4-18-2007

Cellular Planets: Optimizing Planetary Simulations for the Cell Processor

Brent Peckham
Trinity University

Cellular Planets: Optimizing Planetary Simulations for the Cell Processor

Brent Peckham

A departmental honors thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for graduation with departmental honors.

04-18-2007

_____
Thesis Advisor

_____
Department Chair

_____
Associate Vice President for
Academic Affairs

Abstract

The Cell Broadband Engine is the first version of a new type of microprocessor developed jointly between IBM, Toshiba and Sony. The Cell is a multiprocessor that contains nine separate processors that all operate on a shared main memory. This thesis alters C code of planetary simulations that use numeric integrators with gravity as the inter-particle force, and optimizing the simulations to run on the Cell processor so the chip's power can be harnessed. These optimizations are benchmarked and compared to one another in order to see what is necessary to get the most out of the Cell processor. In addition to doing benchmarks for performance the impact of how data structure arrangement affects performance will be looked at and how data structure arrangements can be used to improve the performance.

Cellular Planets: Optimizing Planetary Simulations for the CELL Processor

Brent Peckham
Brent.Peckham@trinity.edu

Advisor: Mark Lewis
mlewis@trinity.edu

Trinity University Department of Computer Science, San Antonio, TX

Acknowledgements

The author wishes to express his appreciation to Dr. Mark Lewis for overseeing and advising on the project, the Trinity University Computer Science Department for providing facilities to work in, and Trinity University for giving the opportunity to pursue such research.

Table of Contents

List of Tables

List of Figures

1 Why Go Cell?

With current technology, it is not economically feasible to create a single processor chip to go any faster than they do now. Because silicon single processor chips increase the heat they release as their speed increases, there is a speed where the heat will become too much and melt the chip. Technology has already reached that point. With current technology, single chip processors cannot maintain a four gigahertz processing speed without burning out after a short period of time. How can this obstacle be overcome? The answer is very simple. Two processors are better than one. Through a parallel processing architecture, computers are able to achieve performances well beyond the four gigahertz limit. How it works is that multiple processors are etched on a single chip and can work in parallel to run operations. While no individual processor achieves great speeds, together their combined power can be massive with certain applications. Due to the potential increase in performance in parallel processors and the lowering prices on the technology, more and more consumer and business products today are shifting towards using the parallel architecture. The Cell Broadband Engine (CBE) takes the parallel processing concept and builds upon it creating a chip that is one of the most powerful parallel processors available to the average consumer today. It is important to mention that eight of the nine processors are only vector units, and not full processors. Unlike other parallel processors such as the Pentium Core Duo and AMD Opteron, these eight processors are not just copies of the original core.

Running a computationally intensive application that can easily be parallelized and split up among eight different processing units would be a good way to find out exactly how much more powerful the CBE can potentially be over other parallel

processors. An application that uses numerical integrators, algorithms used to find the computation of the numerical value of an integral, to form a physics simulation would be a good way of doing this.

Symplectic integrators are a class of numerical integrators that are exact solutions to a discrete Hamiltonian system and are close to the continuum Hamiltonian of interest. They preserve all Poincare invariants and place stringent conditions on the global geometry of the dynamics (8). Symplectic integrators are often used in molecular dynamics and celestial mechanics. In simulations that require the system to be evolved for many dynamical times they are essential. Without the use of a symplectic integrator in such systems, the numerical instabilities would lead to erroneous results.

A common way to code N-body simulations with an integrator of second order accuracy is to use a leap-frog integrator because of its low computational cost for a symplectic integrator compared to other symplectic integrators. There are several advantages to using a leap-frog integrator over other methods. A leap-frog integrator with second order accuracy only needs one force evaluation and one copy of the physical state. In N-body simulations in which the cost of a force evaluation is very expensive, this is very helpful. As stated previously, it is also a symplectic order and preserves Hamiltonian systems' properties. Also in general, since the force field in an N-body simulation is not smooth, higher order does not necessarily provide higher accuracy (8). This is not always true for planetary systems, but there are other reasons why using second order in long planetary simulations is acceptable. These reasons deal with the systems being inherently chaotic, and the simulations being statistical, not predictive, in what they tell us. The following is pseudo-code for the leap-frog integrator:

```
Timestep(t)
{
      for (int i = 0; i < NUM_PARTICLES; i ++)
           for ( int j = i + 1;  j < NUM_PARTICLES; j++)
                calculateDistance (i, j);
                calculateMagnitude(distance, dt); //magnitude of the force
                calculateVelocity(i, j);
       for (i =0; i < NUM_PARTICLES; i++)
            calculatePosition(i);
}
```

The integrator by default will perform roughly $n^2/2$ two way calculations, where n is the number of particles.

Using leap-frog integration C code, which was coded for a single processor chip, optimization for the CBE is attempted in order to see if the CBE does in fact give a performance boost. To help with this project, a software development kit for the CBE is available online and comes with an example Euler integrator that is optimized for the processor. This in part was the reason for testing out a leap integrator, because it is similar code structure-wise. Also, the example Euler code uses constants for force calculations, but gravity will be used for the calculations in this project. Therefore, switching to a leap-frog integrator is appropriate, because it is a much better integrator for doing these types of calculations.

Looking at the bigger picture, it becomes easier to see all the potential uses for the CBE. Since, as of now, the Playstation 3 is the major system powered by the processor, the CBE is used a lot for video games. Being able to solve large and complex calculations quickly has its advantages for use in video games. Things like collision detection, preprocessing for graphics, making large numbers of objects on screen perform actions, etc. can now be done much faster than when being done on other processors.

2 What is Cell?

The Cell Broadband Engine (CBE) is the start of a new line of microprocessors that conform to the Broadband Processor Architecture, a new architecture extending that of PowerPC's. This architecture as well as the Cell was formed as a result of collaboration between Sony, IBM, and Toshiba back in 2001. The CBE is a single-chip microprocessor that contains nine processors operating on shared memory. There are two types of processors, PowerPC Elements (PPEs) and Synergistic Processor Elements (SPEs). A CBE contains one PPE and eight SPEs.

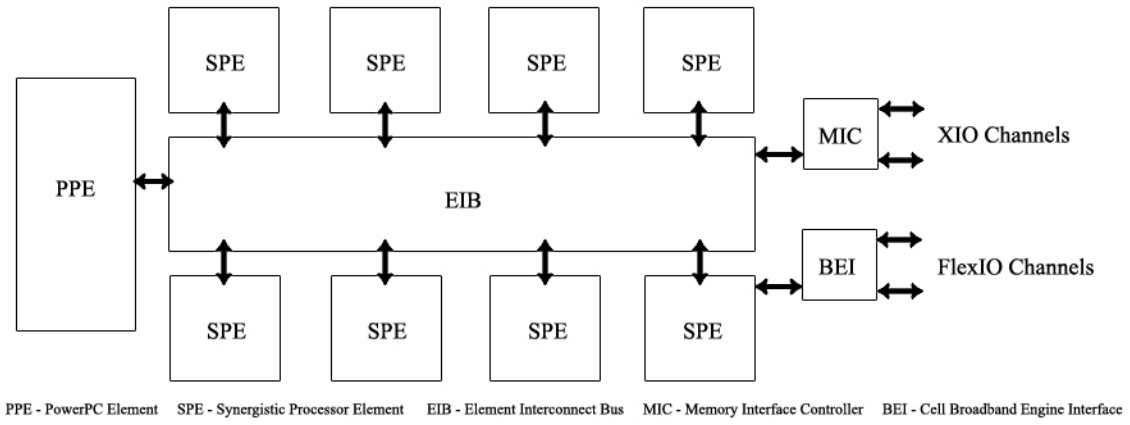

Figure 2.1: Cell Broadband Engine Diagram

The PowerPC Element (PPE)

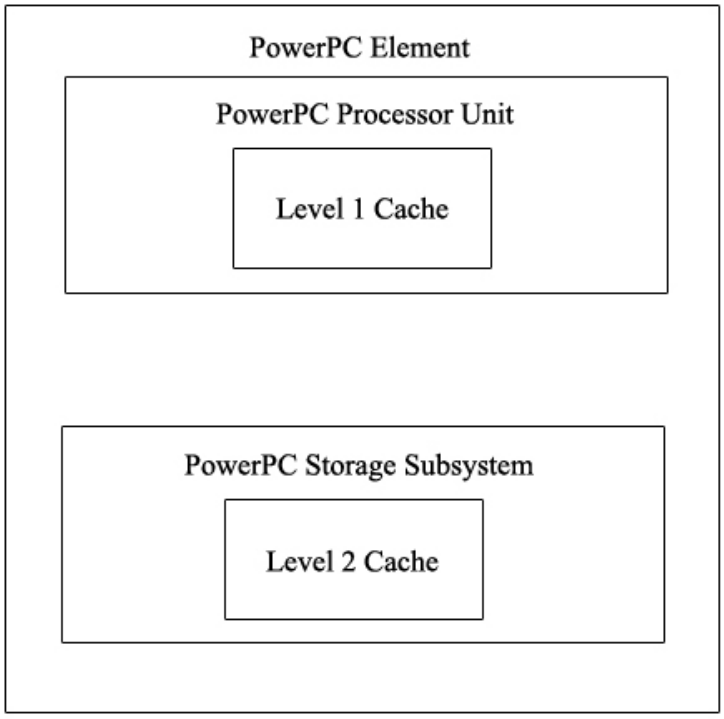

Figure 2.2: PowerPC Element Diagram

The PowerPC Element is a dual-threaded 64-bit RISC processor with a PowerPC Architecture core. It fully complies with the PowerPC Architecture and will also run 32-bit and 64-bit operating systems. The main purposes of the PPE include running an operating system, managing system resources, and control processing. It can run legacy PowerPC software and performs well running system-control code. The PPE also supports the PowerPC Architecture Vector / SIMD (single instruction, multiple data) Multimedia Extension (2). The PPE is made up of two units, the Power Processor Unit (PPU) and the Power Processor Storage Subsystem (PPSS).

The PPU takes care of instruction control and execution. The unit contains the entire set of 64-bit PowerPC registers, 32 128-bit vector registers, a 32-KB level 1 instruction cache, a 32-KB level 1 data cache, an instruction-control unit, a load unit, a store unit, a fixed-point integer unit, a floating-point unit, a vector unit, a branch unit, and a virtual-memory management unit. It can be considered a two-way multiprocessor with shared data flow, because it can run two threads simultaneously. This looks like two separate processing units to the software, and the state for each thread is duplicated. Most non-architected resources (e.g. caches and queues) will be shared by both threads unless the resource is too small or offers a critical system performance boost by not doing so (3).

The PPSS manages PPE memory requests as well as external requests to the PPE from other processors or I/O. It contains a unified 512-KB level 2 instruction and data cache, various queues, and a bus interface unit that takes care of arbitration and pacing on the Element Interconnect Bus (EIB). Memory is viewed as a linear array of bytes ranging from 0 to 2^{64-1}. Each byte is identified by its index (address) and contains a value. One storage access may occur at a time and will be in program order as well. Also, software is able to control the L2 and address-translation caches, which is useful when doing real-time programming (3).

The Element Interconnect Bus

The Element Interconnect Bus is what the PPE and SPEs use to communicate with each other, main storage or I/O. The EIB is a four-ring structure for data and a tree structure for commands. It has an internal bandwidth of 96 bytes per cycle, and the EIB can

support over one hundred outstanding DMA memory requests between the SPEs and main memory (2).

The Synergistic Processor Elements

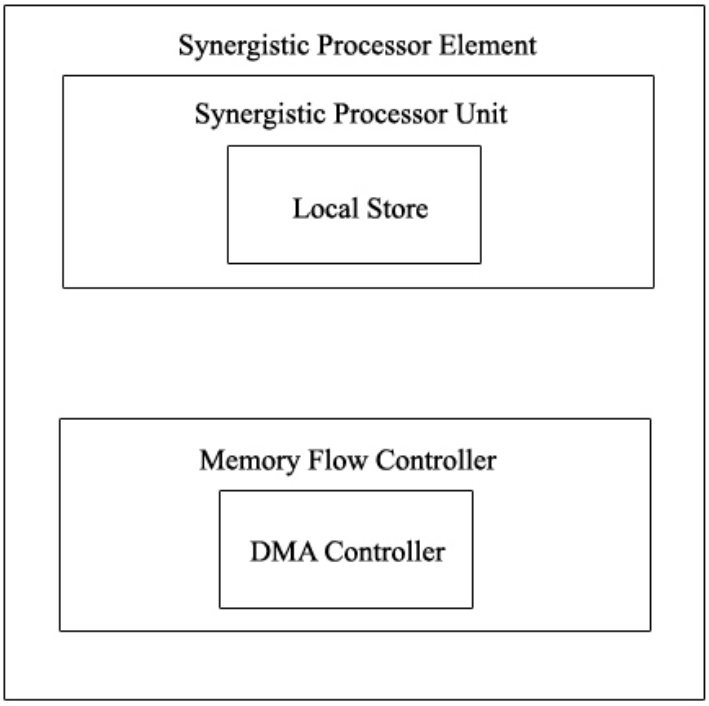

Figure 2.3: Synergistic Processor Element Diagram

The Synergistic Processor Elements are independent processors running individual applications and are optimized for running compute-intensive applications as opposed to operating systems (1). Every SPE contains a 256-KB local store for instructions and data as well as a sizeable (128-bit, 128 entry) unified register file (2). SPEs have full access to the shared memory, including memory-mapped I/O space. These elements are called synergistic because of the high level of dependence between the SPEs and PPEs. The PPE runs the operating system and often the top level control thread of an application,

and the SPEs do a majority of the application processing. SPEs are designed with programming in high-level languages in mind and include an instruction set with SIMD functionality. While use of SIMD is preferred for code run on the CBE, it is not a requirement (3). The SPEs access main memory storage through direct memory access (DMA) commands that go between main storage and private local memory. These DMA transfers access the main storage by using PowerPC effective addresses (2). An SPE is made up of two units: the Synergistic Processor Unit (SPU) and Memory Flow Controller (MFC).

The SPU, like the PPU, is in charge of instruction control and execution, and the unit has a register file containing 128 registers that are 128 bits wide, a unified 256-KB local store, an instruction-control unit, a load unit, a store unit, two fixed-point units, a floating-point unit, and a channel-and-DMA interface. The SIMD instruction set SPUs use is original to the Broadband Processor Architecture. Every SPU has its own program counter and is optimized for running SPE threads spawned by the PPE. The SPU may retrieve data from its own local store or load/save data to and from the local store.

The MFC is the unit that supports DMA transfers through a DMA controller. Programs running on any SPU or the PPE use the MFC's DMA transfers to move data to and from the SPU's local store and main memory. The MCF interfaces the SPU to the EIB and synchronizes operations between the SPU and other processors. The MCF is able to handle DMA transfers because it maintains and processes queues of DMA commands. Even when a DMA command has been queued to the MFC, the SPU can continue to execute instructions. If a DMA-list command is issued, the MCF can execute

a sequence of DMA transfers. Because SPU instructions and DMA commands are executed autonomously, DMA transfers can be scheduled to hide memory latency.

All DMA transfers can be as large as 16 KB in size, but only the SPU associated with the MFC is capable of issuing DMA-list commands. DMA-list commands may represent up to 2048 DMA transfers. The operating system running on the PPE provides each MFC with virtual-memory address translation information, and the system storage attributes follow the page/segment tables of the PowerPC Architecture. While certain privileged software on the PPE can map LS addresses and certain MFC resources to the main storage address space, the aliased memory is not coherent in the system. Also while the local store is shared between DMA read/write operations, load/store operations, and instruction pre-fetch, DMA operations accumulate and can only access the local store at most once every eight cycles. Furthermore instruction pre-fetch delivers at least 17 sequential instructions from the branch target, therefore limiting DMA impact on loads, stores, or program execution time (4).

Coding on the Cell

As stated earlier, the instruction sets for the PPE and SPE are extended versions of the PowerPC instruction set. The PPE and SPE may both execute SIMD instructions, but the instruction sets are different and require programs for the two processors to be compiled by different compilers.

A majority of the Vector/SIMD Multimedia Extension and SPU instructions will operate on vectors; hence vectors are called SIMD operands or packed operands. SIMD processing exploits data-level parallelism, which causes each instruction to operate on a

set of vector elements in parallel. In the PPEs and SPEs, vector registers will hold multiple data elements as one vector. The data paths and registers for SIMD operations are 128-bits wide, which correspond to four-full 32-bit words (5).

Vector/SIMD Multimedia Extension and SPU instruction sets include C-intrinsics. In each instruction set, most intrinsics will have a standard prefix for their names. For example, the intrinsic for the SPU assembly language instruction, add, is spu_add. There are four different categories of intrinsics: specific, generic, composite, and predicate. Specific intrinsics are one-to-one mappings to single assembly language instructions, while generic intrinsics will map to one or more assembly language instructions depending on parameters. Composite intrinsics are those that are constructed from a series of specific or generic intrinsics, and these intrinsics can only be used only on SPEs. Predicate intrinsics are there to evaluate SIMD conditionals, and these intrinsics can be used only on PPEs.

When using a Linux operating system, the main thread of the program runs on the PPE, and that thread can spawn one or more CBE Linux tasks. A CBE Linux task is associated with a number of Linux and SPE threads, threads created to run on a free SPE. All Linux threads in a task will share resources, even access to SPE threads. The two thread types, Linux and SPE, interact with each other through the SPE's local store and can interact indirectly through effective address memory. While waiting for SPE threads, a thread is able to poll or sleep (6).

The operating system defines the policy for choosing an available SPE. The operating system must also prioritize among all the CBE Linux applications running on the system as well as schedule SPE execution separate from the Linux threads. Finally,

the OS is also responsible for runtime loading, parameter passing to SPE programs, SPE events/error notification, and debugger support.

When trying to figure out how to distribute the workload across the nine processors of the CBE, it is important to consider many factors such as processing-load distribution, program structure, program data flow and data access patterns, cost in time and complexity of code and data movement across processors, and cost of loading the bus and bus attachments. There are also two different models for creating programs for the CBE, the PPE-centric and SPE-centric models. The PPE-centric model is one in which the PPE handles the main program and passes tasks off to the SPEs. The PPE will then wait for the SPE results and coordinate them. In the SPE-centric model, a majority of a program's code is handed off to the SPEs, and the PPE mainly acts as a resource manager (7).

3 Methods and Procedures

In order to code for the CBE, Linux must be downloaded and installed on a workstation. A regular PC or Playstation 3 (PS3) may be used for this. The project described in this thesis was done starting on a PC and eventually moved to the PS3. When using a regular PC or laptop, Fedora is the distribution of Linux that is needed in order to install and use the Cell software development kit. The development kit may work on other distributions of Linux, but it is guaranteed to work with Fedora. Before installing the sdk, other dependencies must be installed if they aren't already. After the dependencies are installed, the Cell software development kit can be downloaded and installation initiated. Unfortunately before installation of the sdk can be completed, a multitude of binaries and patches must be downloaded. It took several hours just to get under a dozen small files. After installation, compiling and running code for the CBE is possible. The sdk comes with an emulator, which allows for compiled code to run in a simulated Linux environment that uses the Cell processor. Unfortunately the emulator takes a long time to get up and running, and because it is an emulator, programs do not run in real time. They take significantly longer to finish running than they would on a real Cell. The other and easier alternative to using a PC to create programs for the Cell is to use a PS3. Yellow Dog Linux, a distribution exclusively for the PS3, is available for download and includes all of the libraries for the CBE. Yellow Dog can compile and run Cell code immediately. Since it doesn't use an emulator to run the code, programs will run at real time speeds. Using a PS3 is by far the better of the two options for doing Cell programming.

For this project, Eclipse was the chosen coding environment. Eclipse has plug-ins available that do syntax checking for CBE code. The code created for this project was

one that does an N-body planetary simulation using gravity as the inter-particle force and a leap-frog integrator for the force calculations. The reasoning behind using a leap-frog integrator was that the CBE sdk that was previously mentioned comes with some sample code, which included code that shows the optimization for the CBE of an N-body system using the Euler method. It was helpful to have this for referencing, but that is not to say that the switch from the example Euler code to the code for this project was a simple one. As stated earlier, there was a major difference. The Euler sample code used a constant for the force calculation, while this project's code uses gravity.

This project has five major differing versions of code, and each one marks an important point on the route to code optimization for the CBE. By dividing the code up, benchmarking at various stages of the optimization stages was possible. All different versions of the code created are given in Appendix B.

Basic C Code

The first step of the project was to write up basic code for the leap-frog integrator. This version had nothing special done to it to take advantage of the CBE's capabilities. It was just straightforward C code. This basic version of code allowed for some interesting benchmark tests in addition to providing the baseline for CBE optimizations. It was run on four systems with different chips: Pentium Dual Core, Pentium D, AMD Opteron, and Sony Cell Broadband Engine. This was done to determine whether the C code ran better on a single or multi-processor chip. It was expected to run better on the single-processor chip systems, because code usually runs slower on multi-processor chip systems unless

that code has been optimized for parallel processing. There were no real problems or setbacks encountered in creating this version of code.

Leap Vectored

The basic code was then reorganized to create a version that did not contain actual parallel processing, but instead added Vector/SIMD functionality to the basic code. Note that when "vector" is now referred to, unless specified otherwise, it means something different from the std::vector of C++. A C++ std::vector can have any number of elements, as specified by the programmer. The Cell's SIMD vector registers, however, are 128 bits, so each vector data type holds four 32-byte data values.

In this version of the code, all particles of the system were held in two different 4D vector arrays, one that kept track of particle positions and one that kept track of particle velocities. 4D vectors were custom structs that hold four floats, an X, Y, Z, and W. These 4D vectors were used to simulate the SIMD vectors that are used in later versions of the code. This is why there was a W variable. As stated earlier, the SIMD vector registers have 128 bits. Because of this, bandwidth and memory got wasted. Every vector in the vector array contained four floats, but particles only needed three floats for X, Y, Z positions. The fourth float remained unused and wasted. Something important to note is that instructions used by the Vector/SIMD Multimedia instruction set are 4 bytes long and word-aligned. This resulted in seeing '__attribute__ ((aligned (16)))' after many variables in the code. SIMD instructions to do operations were added throughout the code. For example, add and subtract functions were replaced with calls to vec_add() and vec_sub(). Also values had to be splatted across vectors instead of simple

value assignments to the vector. Therefore, vectors could only have one value directly assigned to all of its elements. Differing values were obtained by applying vector operations to the vector or through other means. Due to the fact the code incorporated the Vector/SIMD Multimedia Extension instruction set, performance results on the CBE were expected to be slightly better than the straight C code version of code.

In this version of the code, the first real issues started to show up. It took awhile to get used to the Vector/SIMD instructions. The fact that vectors' individual elements could not be extracted made it difficult to use prints to do testing and debugging. To print out an individual vector element, the entire vector was typecast and assigned to a float * variable. Individual elements of the float * were printed and reflected what was contained in the associated vector. Another frustration with Vector/SIMD instructions was the painful lack of vector multiply. There were vector adds and subtracts, but the closest thing to multiple among the instructions was vec_madd(), which was multiply add. This multiplied two vectors and added that result to a third one. Because in many cases just a multiply was needed, the added vector was filled with a zero vector. These set backs were minor when compared with issues encountered in the next version of code.

Leap Quad-Vectored

The next version of code took vectoring much further. The 4D vector arrays of position and velocity became just vector float arrays. Furthermore, there were separate vector float arrays for X, Y, and Z values of position and velocity, that brought the total number of arrays to six as opposed to the two used in the previous version of the code. There were two very big advantages to this move. Calculations would be done four

particles at a time as opposed to just one. Since each vector held four values, there were X, Y, and Z values for four different particles in every element of the vector float arrays, position and velocity. The biggest advantage though, was that wasted bandwidth and memory from the last version of code no longer existed. Because the vector float arrays in this version held X, Y, and Z in separate float arrays, there was no wasted W variable. A huge logistics problem appeared in this version of the code. Unless major changes to the code were made from the previous version, particles would only interact with one out of every four instead of every particle interacting with every other. The reason this happened was because the way things were organized. The loops would cause each vector of four particles to interact only once with another vector of four particles as shown in the diagram below.

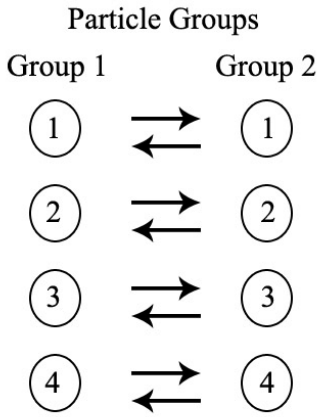

Figure 3.1: Interaction between particles before code alterations

This was corrected by adding another loop inside the already doubly nested loop. This new loop caused two groups of particles to interact with each other four times. Each time they interacted, the second group had particles rotate. This allowed for every particle in group one to interact with every other particle in group two. Checks were also added in to prevent particles from interacting with themselves. The following diagram shows how

the rotation of the second group of particles works to allow interaction between all particles within the two groups.

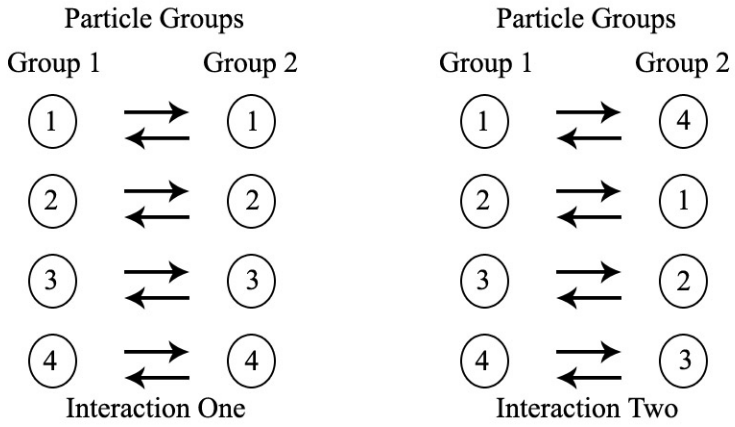

Figure 3.2: Interaction between particles with rotations in place

In order to accomplish these rotations, an extra six vector float variables were also needed. It was uncertain if doing particle calculations in groups of four, would be enough offset the added variables and loop and give it a performance boost.

Leap Single SPE

Finally, it was time to start adding Cell functionality to the code. This next version of code was designed to run on the PPE and one SPE. Making this change required separate .c files for PPE code and SPE code. Also, all key variables associated with this planetary simulation such as position, velocity, and mass have pointers associated with them in a struct called context. In the PPE .c file, the initialization of the position, velocity, and mass vector arrays took place, and the pointers in the context structure were associated with these values. The only other thing that the PPE code does in this version was create and run an SPE thread. Once this was done, the SPE took over

and does the majority of the work. This way of doing things followed the SPE-centric model mentioned in the previous section.

The SPE .c file looked vastly different than any of the previous coding done. This is because SPU specific instructions were used and DMA calls were made. In the SPE part of the code, there were two vector arrays associated with position and two with velocity. There were also two arrays for mass. The vector arrays in this version of code went back to holding X, Y, Z, and W particles for each array element as opposed to three separate arrays for each coordinate variable. Then why were there two arrays associated with position and velocity? Instead of the entire set of particles being loaded onto an SPE at once, it could only take a portion of them. These portions of particles were called blocks. Two blocks of particles were to be loaded in at a time and interact with each other and themselves. Each block was to be loaded into the two vector arrays for position and velocity, and a block would be loaded into each of the two mass arrays. Before any block loading could take place, a DMA call had to be made to load in the context structure. A quadruple-nested loop was what was needed in this version of code to do the force calculations. In the outer loop, the first block of particles was loaded in, and in the second to outer loop, the second block of particles was loaded. The actual force calculations were contained in the two innermost loops. Two versions of the SPE .c file were created. The differences between the two were how the quadruple-nested loop handled the force calculations. In one version, particles did a two-way interaction. When particle A interacted with particle B, particle B would also interact with particle A. The other version of the SPE .c file only contained one-way interactions between particles. The two-way interaction .c file was expected to have faster results than the one way

interaction, but it was still important to have the one-way interaction file for when code was created that ran on multiple SPEs. The two-way interaction version of the code does not translate well to multiple SPEs because it would not distribute across multiple SPEs nicely. The interaction calculations can be viewed as a matrix with the particles forming the rows and columns. The diagonal of the matrix would be the interaction of a particle with itself and these are never calculated. Two-way interactions can be viewed as running through either the upper or lower triangular matrix and applying forces to the particles represented by both the row and the column. This is the approach normally taken on sequential machines because much of the time is spent doing the distance calculations, which are the same regardless of which direction the force is calculated for. However, a triangle is a hard structure to split evenly across multiple processors. The one-way interactions can be viewed as going through all of the cells in the matrix save those on the diagonal. It requires twice as many distance calculations to be performed, but the square structure can be easily divided across processors in a manner that yields good load balance.

It was estimated this version of code would perform better than any of the previous versions regardless of whether the one or two-way interaction .c file was used for the SPE code. The expected increase in performance was a result of this version of code taking advantage of the PPE and SPE. Multiple tests were run using one-way and two-way interactions. Recall that particles are loaded in a block at a time. By changing the number of particles in each block that loaded in, performance was expected to be affected. Tests were run on 500 and 1000 particles per block with a total of 10000

particles in the entire system. Tests were also run on 1000 particles per block in a 1000 particle system.

Leap Multiple SPE

Adding multiple SPE functionality to the single SPE code required few alterations. It would have required a bit of effort if the one way interaction .c file for the single SPE code had not been created. But since it was, all that was need were minor alterations to the PPE .c file to pass particles off to multiple SPEs. Many tests were done using the multiple-SPE code,varying both particles per block and number of SPEs. Using only one SPE was expected to yield performance results similar to the previous SPE code. However, it was uncertain by what factor performance would increase when adding more SPEs and altering the block size. Tests were run on 500, 1000, and 20 particles per block with one to seven SPEs doing the workload. The block size of 20 was chosen because with 20 particles per block, the entire block could fit into the SPEs' registers. It was uncertain how this would change performance results. The reason for using a maximum of seven vector units rather than the eight mentioned earlier in the paper is that only seven are usable to allow room for errors during the manufacturing of the chip. If the chip has a damaged SPE, only having seven SPEs usable prevents any problems this might cause.

The remaining changes to the code that were made and tested all dealt with altering the multiple SPE version of the code. These changes were made in hopes of optimizing the code even further and seeing even more of a performance boost. These

were the final changes made to the leap-frog integrator code with hopes of tapping the CBE's true potential.

The first additional alteration made to the code was to add in double buffering. Double buffering was added in to the code with the idea that a majority of the work being done by the processors was loading and storing to/from main memory and the local stores. Assuming that this was the case, double buffering should have helped improve performance by having another block of particles being loaded in while the first block was being operated on. However if a majority of the work being done was from something other than loads and stores, no gain in performance was expected.

A different change that could be made to the multiple SPE code was loop unrolling. As stated earlier, the CBE likes to do things in groups of four. Therefore the inner loop, where the force calculations take place, had its iterator altered to increment by four. Also, within this loop, the calculations were changed to occur on four particles at a time. These changes caused a small problem though. There were certain instances where a force calculation was not supposed to take place in order to prevent calculating particles against themselves. Extra checks needed to be placed to see if such a situation occurred when the loop was calculating four particles at a time. When such a situation did occur, the force would be calculated using the one particle at a time version of the code. This case should not happen often enough to significantly affect performance. There was still a significant performance boost expected. Another important change in the loop-unrolling version of the code was to change the mass variable into an array of vector floats. Mass used to be an array of floats, but the CBE has an easier time loading and storing vector floats. This change was expected to yield some improvement in the performance as well.

The final thing to do with the multiple SPE code was to combine the double-buffering and loop-rolling alterations into one final piece of code. This version was expected to yield the best performance results out of all versions of code. It is also the conclusion of the attempts to optimize the code for the CBE.

4 Results

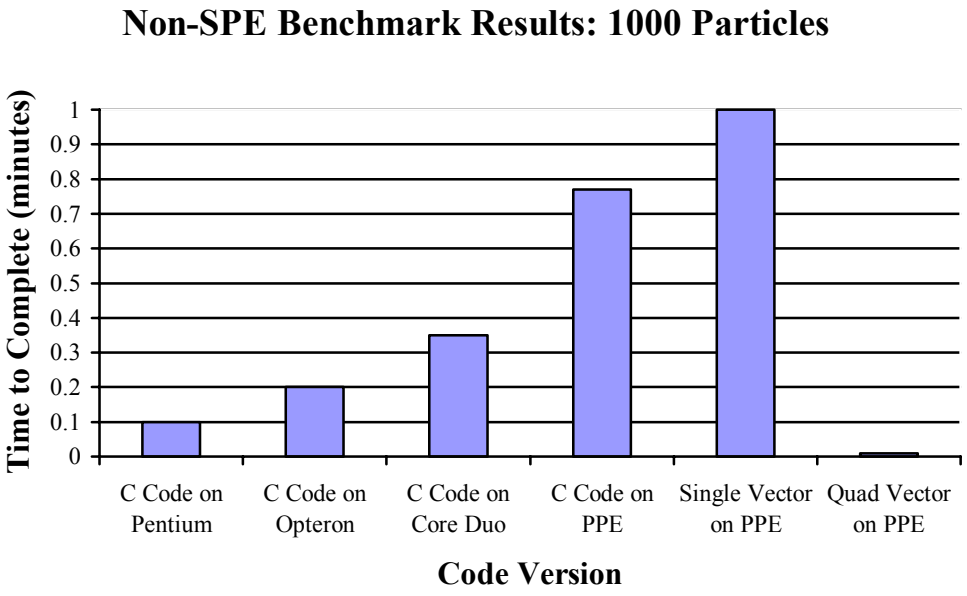

Figure 4.1: Non-SPE Benchmark Results for 1000 Particles

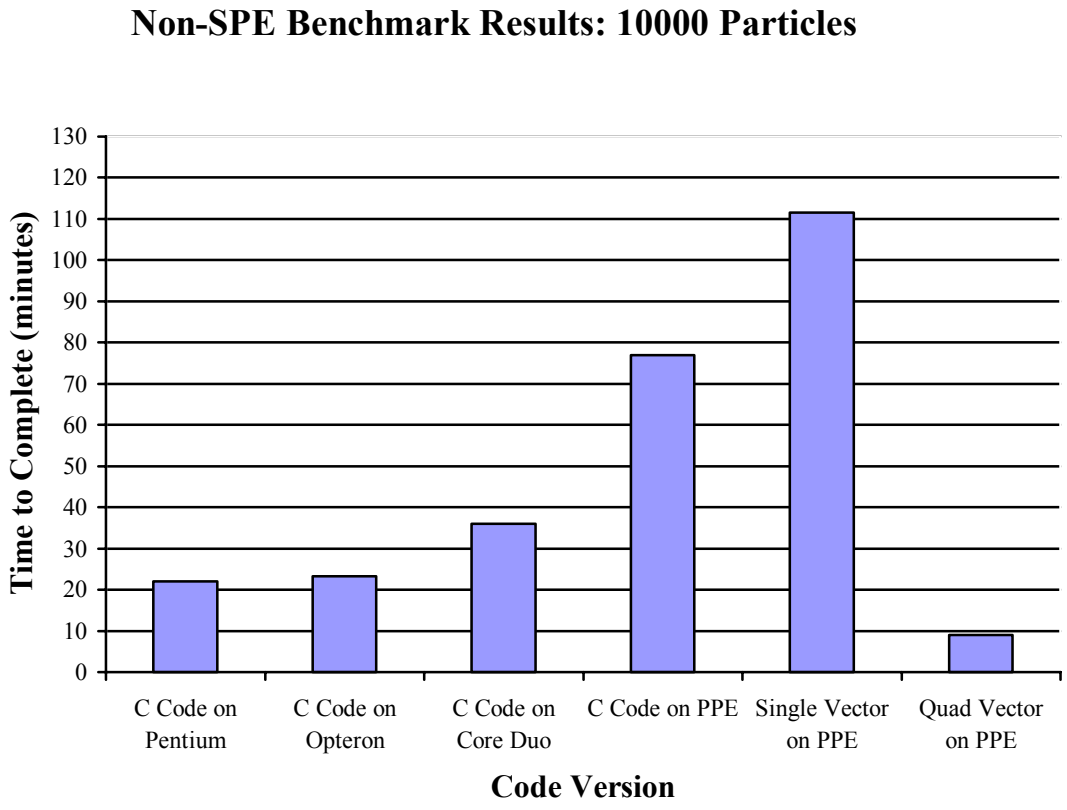

Figure 4.2: Non-SPE Benchmark Results for 10000 Particles

The results in Figures 6 and 7 provide both expected answers as well as a complete surprise. Looking at the results for the basic C code, the graphs reflect what is predicted earlier in the paper. The C code is not written optimized for parallel processing, and thus is expected to run more poorly on the systems with parallel processors. Therefore, the code performs best on the Pentium processor over the Opteron, Core Duo, or CBE PPE.

There were no expectations for the results of the single vector code running on the PPE. It was uncertain whether or not it would perform better than the basic C code on the PPE, but because it was still an early step in the vectoring process, a worse performance time than the basic code would be acceptable. That is exactly what happens. The single vector PPE code is the spike on the graph and has the worst times out of all the Non-SPE benchmark tests.

On the other hand, the results for quad vector running on the PPE are not what were expected at all. As a refresher, the quad vector code calculates the forces of four particles at once. It is assumed that the quad-vector version of the code would have an improved performance over the single vector, but the fact that the performance time improves by a factor of more than ten is shocking. This improvement makes it fastest of all non-SPE benchmark tests by a large margin. At this point in the testing phase, the SPE code hasn't been tested yet. The quad vectoring improves the performance of the code greatly. It will be interesting to see what kind of results the SPE testing will yield.

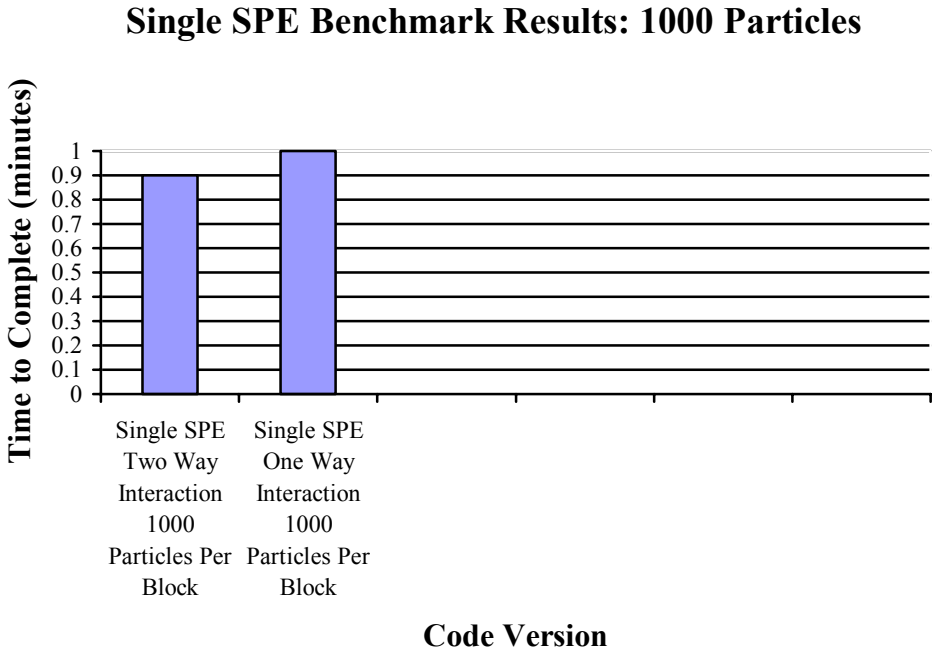

Figure 4.3: Single SPE Benchmark Results for 1000 Particles

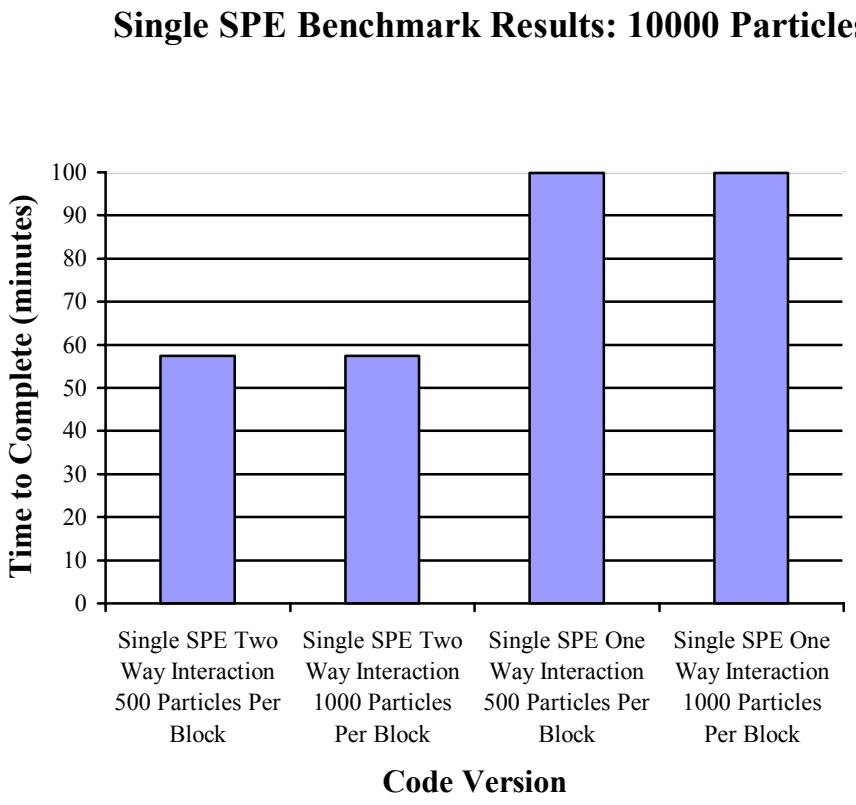

Figure 4.4: Single SPE Benchmark Results for 10000 Particles

The code that runs on a single SPE has four different versions. One way and two way interactions between particles were tested with using 500 and 1000 particles per block, thus creating four versions. It is interesting to note that at this stage the block size does not seem to make much of a difference on a single SPE, considering that tests with both block sizes give [?] roughly the same times for all four runs. While the block size may not matter on a single SPE, it definitely has a much larger impact when the code is run on multiple SPEs.

Looking at Figure 8, one can see that the performance difference is small between the one and two way interaction on a single SPE and using 1000 particles. There is only a 10% difference between the two runs. Figure 9 however, shows a much more notable difference in times, closer to 100%. It is more expected to see a 100% decrease in speed than 10% when changing from a two-way interaction to a one-way interaction. This is because twice as many interactions are occurring. There is not enough information to form a conclusion, but it might be possible that something other than numerics are dominating the time at 1000 particles though they do seem to be showing the factor of 100 scaling relation.

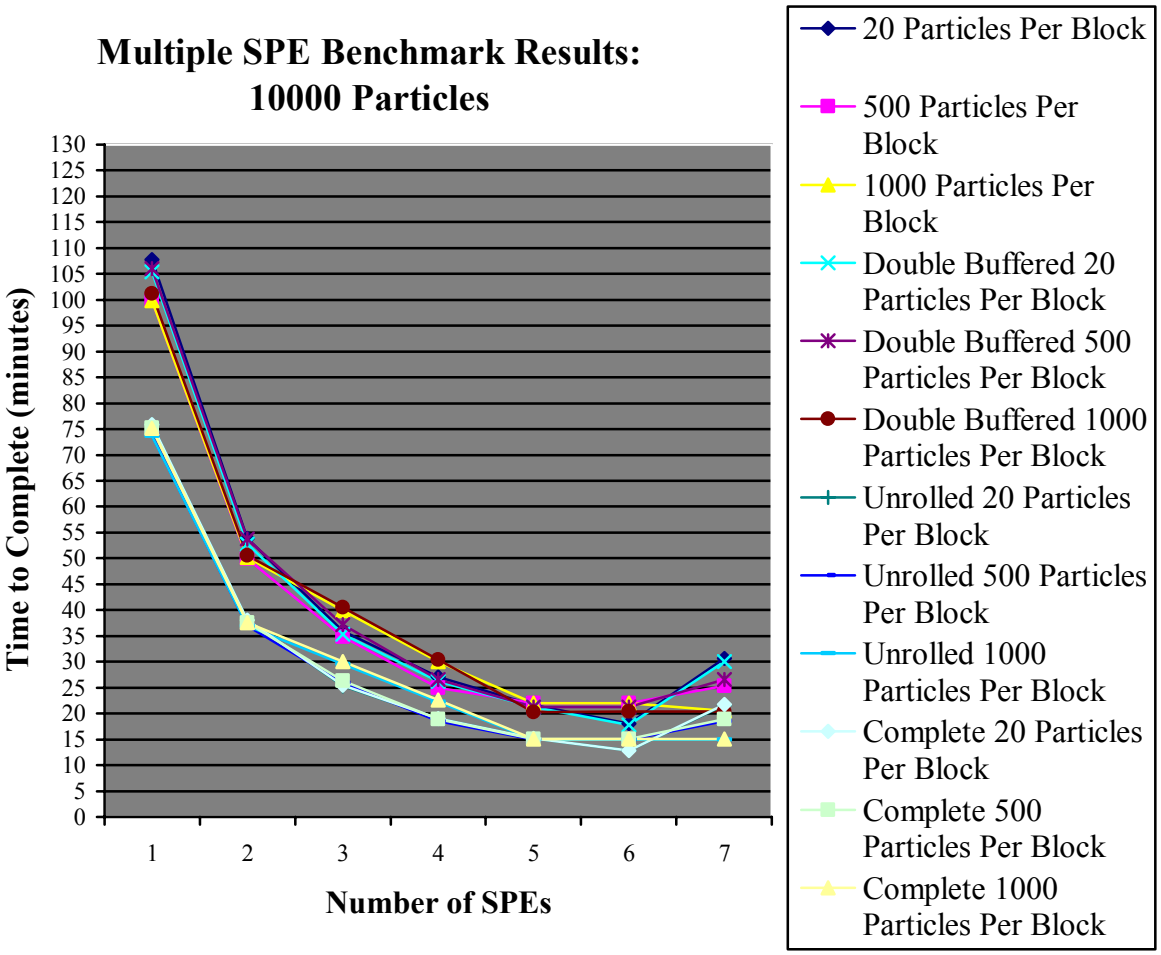

Figure 4.5: Multiple SPE Benchmark Results for 10000 Particles

With the multiple-SPE versions of code, more than just block size is looked at. Tests are run on the code that also alters the number of SPEs being used. Furthermore, different versions of the same code are run to test to see if adding in loop unrolling or double buffering will affect performance. In the multiple-SPE code versions, all tests are performed using 10000 particles for the system.

Figure 10 shows all of the benchmark results for the different versions of multiple-SPE code. As one can see, there are mainly two groupings of results. The default and double buffered versions of SPE code seem to yield similar results, as do the versions that use loop unrolling and a combination of loop unrolling and double

buffering. Block size also now affects performance to a larger degree than on the single SPE code, but the effects it has is still not that important.

The default version of code is tested using block sizes of 20, 500, and 1000. For runs with block sizes of both 20 and 500, the performance time improves or roughly stays the same as more SPEs are being used, until the seventh SPE is added. Seven SPEs tend to perform worse than six, especially with a block size of 20. However, on the runs that use a block size of 1000, the decrease in performance is usually negligible when compared to the other two block sizes. The particles per block run performs slower than the 500 particles per block run except for when five and six SPEs are in use, and it performs slower than the 1000 particles per block run except for when three through six SPEs are used. The 500 particles per block run performs the same as the 1000 particles per block run when one SPE is being used. This is essentially a repeat of the single-SPE run mentioned earlier. After increasing the SPEs used to two, the 500 particles per block run performs better than the 1000 particles per block run up until the use of five SPEs. At this point the two runs become close to yielding the exact same result, and when seven SPEs are used, the 1000 particles per block run outperforms the 500 particles per block run. Out of all three runs, the best results are yielded by the 20 particles per block run using six SPEs. This has some interesting implications. Because the block size is only 20, that means there will be a much greater number of loads and stores than in the 500 and 1000 particles per block runs. For the most the 20 particles per block runs performance isn't much worse than the other two, and when it uses six SPEs, it yields the best performance.

The double buffered version of code yields similar results to the normal version of code, but there is an important difference to notice. The 20 particles per block run sees a slight improvement in performance when using any number of SPEs, while the 500 particles per block and 1000 particles per block runs see a small decrease in performance when using any number of SPEs. These results were not expected when the tests were initially run, but after some thought they make sense. Double buffering is beneficial during loading and storing. It cuts down the time it takes, improving performance, and because the 20 particles per block run has a much greater number of loads and stores, it was the one to get the benefits of double buffering. And like with the default version of code, the 20 particles per block run using six SPEs yields the best results.

When looking at the benchmarks for the version of code that uses unrolled loops, the overall improvement in performance is easily noticed. Even when using only one SPE, the unrolled code performs roughly 25% better than both the default and double buffered single SPE code. Once again though, a similar performance curve to the previous to versions of code is seen when using the different block sizes and various numbers of SPEs. Use of seven SPEs still shows a decrease in performance. The fastest benchmark of the unrolled loop version still belongs to the 20 particles per block run using six SPEs. This version's benchmark is the fastest of all the multiple SPE code versions tested. The version that uses loop unrolling with double buffering also yields the same result in this situation.

The final version of code that uses both double buffering and unrolled loops is very similar to the unrolled loop version. In fact, the 20 particles per block run yields almost the exact same results. There is a slight decrease in performance seen with the 500

and 1000 particles per block, but that is expected after looking at the results of just the double buffered version's performance. It slowed down the 500 and 1000 particles per block run a little bit there, and this is reflected in this version of code. However the 20 particles per block run does not see the performance boost it saw in the double buffered code version. Apparently, the improvement received from double buffering does not mean much when it is paired with loop unrolling. The gap between performance times of the varying block sizes in this final version of code is lessened except when using five or six SPEs. What this means is that in the end, block size does not have a significant impact on performance time.

The curve in Figure 10 shows the times for each run improving rapidly as the number of SPEs increase, but improvement starts to slow as the curve approaches five SPEs. The greater the number of SPEs that are in use, the fewer number of blocks each SPE needs to handle. This is where the boost in performance time comes in. However as the number of SPEs being used increases, the factor of decrease of blocks that each SPE needs to handle becomes smaller as well. For example, in the 1000 block run with one SPE, the one SPE handles ten blocks. When this same run uses two SPEs, each SPE handles five blocks. The number of blocks each SPE handles decreases by a factor of two from the last step. However when three SPEs are used, one SPE handles four blocks and the rest handle three. This decrease is far less than a factor of two and becomes less as the number of SPEs used increases.

The big mystery, though, is why do seven SPEs perform so much worse than six SPEs with the exception of the 1000 particles per block runs? There is no definite answer for this one, but when looking at the block distribution of the three block sizes at seven

SPUs, a difference shows up between the 1000 particles per block and the other two. This difference is that the number of blocks that most of the SPEs handle using seven SPEs for the 1000 particles per block run is the same as when using five and six SPEs. With the 20 and 500 particles per block runs, the number of blocks each SPE handles decreases when adding the seventh SPE. One would think that this would further improve performance, but perhaps the work is stretched too thin over the seven SPEs or the traffic on the bus hits some critical level. There's also the possibility that the further decreasing of block size has left a bottleneck somewhere. More extensive testing is needed to determine the cause of this phenomenon.

Why are the times almost equivalent for the runs using 500 and 1000 particle particles per block with five and six SPEs? The reason is, once again, because of block distribution among the SPEs. In both cases most of the SPEs handle the same number of blocks when using five and six SPEs. The 20 particles per block does not however, and that is why there is a larger discrepancy in time results during those runs.

So why do the 20 and 500 block size benchmarks tend to outperform the 1000 particles per block benchmarks? One possibility might be SPE register optimization. The SPE register spaces may fit 500 and 20 particles better than 1000. Furthermore with a larger number of overall blocks, smoother distribution and better scaling may occur. This implies that communication is not the primary issue for speed. More time is spent doing calculations than loads and stores. The increased number of overall blocks will not significantly slow things down.

However, the biggest issue and question is why does the quad vector code that does not even use the SPEs beat the complete version of the multiple SPE code? There

are several possible explanations other than the CBE and its SPEs do not deliver the desired benefits, and all of these explanations deal with insufficient optimization and testing. The most likely cause of the time difference is a lack of good C compilers for the SPE. The chip used for the PPE has been around for a while and therefore has C compilers that produce fast code, but the SPEs are brand new. It might be a while before they see fast compilers for the C language, and in the meantime, coding in assembly might be necessary to get the most out of the SPEs in terms of performance. Thread synchronization between SPEs and PPEs has also not been explored yet. As a result a new SPE thread spawns on every time step. How this would affect performance though, is unknown. It is clear by now that the performance problem in the SPE code is occurring somewhere in the calculations. There may be a way to increase the calculation's efficiency either by changing simply changing the logic behind the code or by altering data structures in the code further. There is also a difference between the data structures of the quad vector code and SPE code. The quad vector code uses three separate vectors to hold the position and velocity of particles. The reasoning behind this is to remove the wasting of memory found in other versions of code. Vector floats store four float values. Position and velocity only have three values for their coordinates. By storing these values in a vector float, the fourth space goes unused and is wasted. The quad vector code eliminates this waste. This change possibly has a significant impact on performance time, but not likely a factor of ten. Nothing is certain without testing.

If more time had been available, there were many tests that would have been nice to run on the code. Altering the multiple SPE code to use three vector float arrays a piece for position and velocity would be one of the first things to try. One of the possible

reasons why the Euler code does not bother with doing this is that the code is limited by communication and not the force calculations. Because each vector needs a separate DMA to pull data across to/from the PPE, it probably is not worth having six DMAs for the x, y, z, vx, vy, and vz, vectors instead of two for the position and velocity vectors in the Euler code. Since the leap-frog code is limited by the calculations, adding the extra vectors may be beneficial. Further testing would also provide a good opportunity to spend more time on breaking down the code. This would allow it to be examined more carefully and find out exactly what parts of the code are causing the slow-downs in the overall performance. This would perhaps result in further optimization and improved performance. It would also be interesting to test the code using a much higher number of particles. How the CBE handles a number much larger than 10000 particles would be interesting. Seeing how the performance scales between the versions of code using a larger number of particles versus 10000, could show that the 10000 particle versions do not take full advantage of the Cell. Finally, it would have been beneficial to test out a large structure on the CBE, such as a binary tree. Looking at a structure that may not be small enough to fit on to the SPEs would give an even better idea of some of the issues and changes that are required to alter C code to optimized CBE code. A large data structure would really require one to think about getting the most out of space and memory due to the CBE's limited resources and capabilities.

5 Conclusion

The only solid conclusion that can be drawn from this work at this point is that working with the Cell processor is a challenging endeavor given the current state of tools for the platform. The benchmark results from this project are very interesting. They did not come out as expected. Much better benchmarks were expected of the multiple SPE tests. It was assumed multiple SPE benchmarks would by far out perform the tests done on code that was run on the PPE. While the PPE is technically part of the Cell processor, it is nothing more than a slightly altered PowerPC processor. This does not mean that the Cell is not a powerful processor. As stated earlier, there is a very high possibility that the main reason one of the PPE runs outdid the best SPE run is due to lack of good C compilers on the SPE. More testing and research is necessary in order to figure out how to bring out the Cell's real power. There is not enough time to do all this with one college thesis.

Further research would be worthwhile, and the field of computer science is still very much alive and evolving. The Cell is just one piece of new, cutting edge technology that has recently come out in the field. With more advanced processors coming out that requires people to think outside the box in order to achieve greater results, better tools and languages, such as Sun's Fortress and IBM's X10 languages, are needed. In order for these improvements to happen, more computer scientists that can dedicate time and effort to such research are also required. Right now is a great time to get into the field of computer science. The Cell and the research from this project help reaffirm that.

6 References

1. IBM, “Cell Broadband Engine Programming Tutorial”, IBM.com, http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788/$file/CBE_Tutorial_v2.0_15December2006.pdf, 13.

2. IBM, “Cell Broadband Engine Programming Tutorial”, IBM.com, http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788/$file/CBE_Tutorial_v2.0_15December2006.pdf, 17.

3. IBM, “Cell Broadband Engine Programming Tutorial”, IBM.com, http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788/$file/CBE_Tutorial_v2.0_15December2006.pdf, 19.

4. IBM, “Cell Broadband Engine Programming Tutorial”, IBM.com, http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788/$file/CBE_Tutorial_v2.0_15December2006.pdf, 20.

5. IBM, “Cell Broadband Engine Programming Tutorial”, IBM.com, http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788/$file/CBE_Tutorial_v2.0_15December2006.pdf, 21.

6. IBM, “Cell Broadband Engine Programming Tutorial”, IBM.com, http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788/$file/CBE_Tutorial_v2.0_15December2006.pdf, 23.

7. IBM, “Cell Broadband Engine Programming Tutorial”, IBM.com, http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788/$file/CBE_Tutorial_v2.0_15December2006.pdf, 25.

8. Neal Katz, George Lake, Neal Katz, and Joachim Stadel, “Time stepping N-body Simulations”, arXiv:astro-ph/9710043. Volume 1. October 3, 1997.

Appendix A: Results

	Pentium	Opteron	Core Duo	PPE	Single Vector	Quad Vector
1000 Particles	0m06s	0m12s	0m21s	0m46s	0m46s	0m01s
10000 Particles	22m00s	23m18s	36m00s	76m54s	111m37s	8m58s

Table A.1: Non-SPE Benchmark Results

	Two-way Interaction with 500 Particles Per Block	Two-way Interaction with 1000 Particles Per Block	One-way Interaction with 500 Particles Per Block	One-way Interaction with 1000 Particles Per Block
1000 Particles	-	0m54s	-	1m01s
10000 Particles	57m26s	57m27s	99m54s	99m51s

Table A.2: Single SPE Benchmark Results

Number of SPEs	Default with 20 Particles per Block	Default with 500 Particles per Block	Default with 1000 Particles per Block	Double Buffered with 20 Particles per Block	Double Buffered with 500 Particles per Block	Double Buffered with 1000 Particles per Block
1	107m42s	99m54s	99m55s	105m25s	106m01s	101m13s
2	53m54s	49m58s	50m13s	52m46s	53m04s	50m36s
3	36m02s	35m00s	40m00s	35m17s	37m10s	40m30s
4	27m09s	25m03s	30m01s	26m26s	26m35s	30m24s
5	21m38s	20m03s	20m03s	21m10s	21m18s	20m18s
6	18m11s	20m03s	20m03s	17m48s	21m18s	20m19s
7	30m41s	25m51s	20m30s	30m02s	26m35s	20m22s

Number of SPEs	Unrolled Loop with 20 Particles per Block	Unrolled Loop with 500 Particles per Block	Unrolled Loop with 1000 Particles per Block	Complete with 20 Particles per Block	Complete with 500 Particles per Block	Complete with 1000 Particles per Block
1	75m51s	73m48s	73m45s	75m55s	75m11s	75m08s
2	37m58	36m56s	36m54s	38m02s	37m38s	37m38s
3	25m23s	25m53s	29m33s	25m25s	26m22s	30m06s
4	19m02s	18m31s	22m21s	19m3s	18m52s	22m37s
5	15m15s	14m51s	14m50s	15m16s	15m07s	15m07s
6	12m49s	14m50s	14m51s	12m50s	15m07s	15m07s
7	21m38s	18m31s	14m51s	21m39s	18m52s	15m09s

Table A.3: Multiple SPE Benchmark Results

Appendix B: Code

Step 1: Basic C Code

leap.h

```
#ifndef PARTICLE_H_
#define PARTICLE_H_

typedef struct{
    double x, y, z;
    double vx, vy, vz;
    double m;
}particle;
#endif /*PARTICLE_H_*/
```

leap.c

```
#include <math.h>
#include"leap.h"

#define NUM_PARTICLES 10000
#define END_OF_TIME 6.28

int main(int argc,char **argv)
{
      int i, j;
      double time;
      double dt =.01;
      particle p[NUM_PARTICLES];

      for(i = 0; i < NUM_PARTICLES; i++){
           p[i].x = 0.0;
           p[i].vx = 0.0;
           p[i].y = 0.0;
           p[i].vy = 0.0;
           p[i].z = 0.0;
           p[i].vz = 0.0f;
           p[i].m = 1.0f;
       }
      for(i = 1; i < NUM_PARTICLES; i++){
            p[i].x = i;
            p[i].vy = 1.0/sqrt(i);
            p[i].m = 1e-4;
      }
      for(time = 0; time < END_OF_TIME; time+=dt){
           for(i = 0; i < NUM_PARTICLES; i++){
                for(j = i+1; j < NUM_PARTICLES; j++){
                     double dx = p[i].x - p[j].x;
                     double dy = p[i].y - p[j].y;
                     double dz = p[i].z - p[j].z;
                     double dist = sqrt(dx*dx+dy*dy+dz*dz);
                     double mag = dt/(dist*dist*dist);
```

```
                    p[j].vx += p[i].m*mag*dx;
                    p[j].vy += p[i].m*mag*dy;
                    p[j].vz += p[i].m*mag*dz;
                    p[i].vx -= p[j].m*mag*dx;
                    p[i].vy -= p[j].m*mag*dy;
                    p[i].vz -= p[j].m*mag*dz;
                }
            }
            for(i = 0; i < NUM_PARTICLES; i++){
                p[i].x += p[i].vx * dt;
                p[i].y += p[i].vy * dt;
                p[i].z += p[i].vz * dt;
            }
        }

        return 0;

}
```

Step 2: Leap Vectored (Single Instruction Multiple Data with Array of Structures)

leap.h
See Step 1.

leap_simd_aos.c

```
#include <vec_literal.h>
#include <math.h>
#include <sum_across_float4.h>
#include "leap.h"

vec4D pos[PARTICLES] __attribute__ ((aligned (16)));
vec4D vel[PARTICLES] __attribute__ ((aligned (16)));
float mass[PARTICLES] __attribute__ ((aligned (16)));
float dt __attribute__ ((aligned (16))) = 0.01f;
vector float zero = VEC_SPLAT_F32(0.0f);
vector float one = VEC_SPLAT_F32(1.0f);

vector float reciprocalSquareRoot(vector float a)
{
        vector float oneHalf = VEC_SPLAT_F32(0.5f);
        vector float estimate = vec_rsqrte(a);
        vector float estimateSquared = vec_madd(estimate, estimate, zero);
        vector float halfEstimate = vec_madd(estimate, oneHalf, zero);

        return vec_madd(vec_nmsub(a, estimateSquared, one), halfEstimate, estimate);

}

vector float reciprocal(vector float a)
{
```

```
        vector float estimate= vec_re(a);

        return vec_madd( vec_nmsub( estimate, a, one ), estimate, estimate );
}

vector float divide(vector float a, vector float b)
{
        return vec_madd(a, reciprocal(b), zero);
}

vector float squareRoot(vector float a)
{
        return vec_madd( a, reciprocalSquareRoot(a), zero);
}

int main(int argc,char **argv)
{
        int i, j;
        float time;
        int step = 0;
        vector float dt_v;
        vector float *pos_v, *vel_v;
        vector float temp = zero;
        vector float inc = VEC_SPLAT_F32(1.0f);
        vector float total = VEC_SPLAT_F32(0.0f);
        vector float half = VEC_SPLAT_F32(0.5f);

        for(i = 0; i < PARTICLES; i++){
              pos[i].x = 0.0f;
              vel[i].x = 0.0f;
              pos[i].y = 0.0f;
              vel[i].y = 0.0f;
              pos[i].z = 0.0f;
              vel[i].z = 0.0f;
              pos[i].w = 0.0f;
              vel[i].w = 0.0f;
              mass[i] = 1.0f;
        }
        for(i = 1; i < PARTICLES; i++){
              pos[i].x = (float) i;
              vel[i].y = 1/sqrt((float) i);
              mass[i] = 1e-4f;
        }
        pos_v = (vector float *)pos;
        vel_v = (vector float *)vel;
        dt_v = vec_splat(vec_lde(0, &dt), 0);
        for(time = 0; time < END_OF_TIME; time+= dt){
              for(i = 0; i < PARTICLES; i++){
                    for(j = i+1; j < PARTICLES; j++){
                          vector float forceCalc = vec_sub(pos_v[i],pos_v[j]);
                          temp = vec_madd(forceCalc, forceCalc, zero);
                          float dist __attribute__ ((aligned (16)))= sqrt(_sum_across_float4(temp));
                          vector float dist_v = vec_splat(vec_lde(0, &dist), 0);
                          vector float mag = divide(dt_v, vec_madd(vec_madd(dist_v, dist_v, zero), dist_v,
                                                zero));
                          float mi __attribute__ ((aligned (16)))= mass[i];
```

```
                    float mj __attribute__ ((aligned (16)))= mass[j];
                    vector float mass_vi = vec_splat(vec_lde(0, &mi), 0);
                    vector float mass_vj = vec_splat(vec_lde(0, &mj), 0);

                    vel_v[j] = vec_madd(vec_madd(mass_vi, forceCalc, zero), mag ,vel_v[j]);
                    vel_v[i] = vec_sub(vel_v[i], vec_madd(vec_madd(mass_vj, forceCalc, zero), mag
                                ,zero));
                }
            }
            for(i = 0; i < PARTICLES; i++)
                pos_v[i] = vec_madd(vel_v[i], dt_v, pos_v[i]);
        }

    return 0;
}
```

Step 3: Leap Quad-Vectored (Single Instruction Multiple Data with Structure of Arrays)

leap.h
See Step 1.

leap_simd_soa.c

```
#include <math.h>
#include <vec_literal.h>
#include <sum_across_float4.h>
#include "leap.h"

vector float pos_x[PARTICLES/4], pos_y[PARTICLES/4], pos_z[PARTICLES/4];
vector float vel_x[PARTICLES/4], vel_y[PARTICLES/4], vel_z[PARTICLES/4];
vector float mass[PARTICLES/4];
float dt __attribute__ ((aligned (16))) = 0.01f;
vector float zero = VEC_SPLAT_F32(0.0f);
vector float one = VEC_SPLAT_F32(1.0f);

vector float reciprocalSquareRoot(vector float a)
{
      vector float oneHalf = VEC_SPLAT_F32(0.5f);
      vector float estimate = vec_rsqrte(a);
      vector float estimateSquared = vec_madd(estimate, estimate, zero);
      vector float halfEstimate = vec_madd(estimate, oneHalf, zero);

       return vec_madd(vec_nmsub(a, estimateSquared, one), halfEstimate, estimate);

 }

vector float reciprocal(vector float a)
{
      vector float estimate= vec_re(a);

      return vec_madd( vec_nmsub( estimate, a, one ), estimate, estimate );
```

```
}

vector float divide(vector float a, vector float b)
{
    return vec_madd(a, reciprocal(b), zero);
}

vector float squareRoot(vector float a)
{
    return vec_madd( a, reciprocalSquareRoot(a), zero);
}

int main()
{
    int i, j, k;
    float time;
    int step = 0;
    int fourCount = 1;
    vector float dt_v;
    vector float total = VEC_SPLAT_F32(0.0f);
    vector float half = VEC_SPLAT_F32(0.5f);
    vector unsigned char rotate = (vector unsigned char) {4,5,6,7,8,9,10,11,12,13,14,15,0,1,2,3};

    for(i = 0; i < PARTICLES/4; i++){
        pos_x[i] = VEC_SPLAT_F32(0.0f);
        vel_x[i] = VEC_SPLAT_F32(0.0f);
        pos_y[i] = VEC_SPLAT_F32(0.0f);
        vel_y[i] = VEC_SPLAT_F32(0.0f);
        pos_z[i] = VEC_SPLAT_F32(0.0f);
        vel_z[i] = VEC_SPLAT_F32(0.0f);
    }
    pos_x[0] = VEC_LITERAL(vector float, 0.0f, 1.0f, 2.0f, 3.0f);
    vel_y[0] = VEC_LITERAL(vector float, 0.0f, 1.0f, 1/sqrt(2), 1/sqrt(3));
    mass[0] = VEC_LITERAL (vector float, 1.0f, 1e-4f, 1e-4f, 1e-4f);
    for(i = 1; i < PARTICLES/4; i++){
        pos_x[i] = VEC_LITERAL (vector float, (float) 0+(fourCount*4), (float) 1+(fourCount*4), (float)
                                2+(fourCount*4), (float) 3+(fourCount*4));
        vel_y[i] = VEC_LITERAL (vector float, (float) 1/sqrt(0+(fourCount*4)), (float)
                                1/sqrt(1+(fourCount*4)), (float) 1/sqrt(2+(fourCount*4)), (float)
                                1/sqrt(3+(fourCount*4)));
        fourCount++;
    }

    for(i = 1; i < PARTICLES/4; i++)
        mass[i] = VEC_SPLAT_F32(1e-4f);
    dt_v = vec_splat(vec_lde(0, &dt), 0);
    for (time=0; time<END_OF_TIME; time += dt) {
        for (i = 0; i<PARTICLES/4; i++) {
            for(j = i; j < PARTICLES/4; j++){
                vector float pxj = pos_x[j];
                vector float pyj = pos_y[j];
                vector float pzj = pos_z[j];
                vector float mj = mass[j];
                vector float axj = zero;
                vector float ayj = zero;
                vector float azj = zero;
```

```
                vector float dx, dy, dz, dist, mag;

                for(k = 0; k < 4; k++){

                    pxj = vec_perm(pxj, pxj,  rotate);
                    pyj = vec_perm(pyj, pyj,  rotate);
                    pzj = vec_perm(pzj, pzj,  rotate);
                    mj = vec_perm(mj, zero, rotate);
                    axj = vec_perm(axj, zero, rotate);
                    ayj = vec_perm(ayj, zero, rotate);
                    azj = vec_perm(azj, zero, rotate);
                    if(i != j || k !=3){
                        dx = vec_sub(pos_x[i], pxj);
                        dy = vec_sub(pos_y[i], pyj);
                        dz = vec_sub(pos_z[i], pzj);
                        dist = squareRoot(vec_madd(dx, dx, vec_madd(dy, dy, vec_madd(dz, dz,
                                    zero))));
                        mag = divide(dt_v, vec_madd(vec_madd(dist, dist, zero), dist, zero));
                        axj = vec_madd(vec_madd(mass[i], dx, zero), mag ,axj);
                        ayj = vec_madd(vec_madd(mass[i], dy, zero), mag ,ayj);
                        azj = vec_madd(vec_madd(mass[i], dz, zero), mag ,azj);
                    }
                }
                vel_x[j] = vec_add(vel_x[j], axj);
                vel_y[j] = vec_add(vel_y[j], ayj);
                vel_z[j] = vec_add(vel_z[j], azj);
                }
            }
            for(i = 0; i < PARTICLES/4; i++){
                pos_x[i] = vec_madd(vel_x[i], dt_v, pos_x[i]);
                pos_y[i] = vec_madd(vel_y[i], dt_v, pos_y[i]);
                pos_z[i] = vec_madd(vel_z[i], dt_v, pos_z[i]);
            }

        }

 return 0;

}
```

Step 4a: Leap Single SPE (Two-Way Interaction)

leap.h

```
#ifndef _LEAP_H_
#define _LEAP_H_

#define END_OF_TIME   6.28
#define PARTICLES     10000

typedef struct {
    float x, y, z, w;
} vec4D;

#endif /* _LEAP_H_ */
```

particle.h

```
#include "leap.h"

typedef struct {
      int particles;  // number of particle to process
      vector float *pos_v;    // pointer to array of positions vectors
      vector float *vel_v;    // pointer to array of velocity vectors
     float *mass;   // pointer to array of mass vectors
      vector float force_v;   // force vector
     float dt;                // current step in time
} context;
```

euler_spe.c

```
#include <stdio.h>
#include <libspe.h>
#include "particle.h"

vec4D pos[PARTICLES] __attribute__ ((aligned (16)));
vec4D vel[PARTICLES] __attribute__ ((aligned (16)));
vec4D force __attribute__ ((aligned (16)));
float mass[PARTICLES] __attribute__ ((aligned (16)));
float dt = 0.01f;

extern spe_program_handle_t particle;

int main()
{
      int status, i;
      speid_t spe_id;
     context ctx __attribute__ ((aligned (16)));

     for(i = 0; i < PARTICLES; i++){
           pos[i].x = 0.0f;
           vel[i].x = 0.0f;
           pos[i].y = 0.0f;
           vel[i].y = 0.0f;
           pos[i].z = 0.0f;
           vel[i].z = 0.0f;
           pos[i].w = 0.0f;
           vel[i].w = 0.0f;
           mass[i] = 1.0f;
     }
     for(i = 1; i < PARTICLES; i++){
          pos[i].x = (float) i;
          vel[i].y = 1/sqrt((float) i);
          mass[i] = 1e-4f;
     }
     ctx.particles = PARTICLES;
     ctx.pos_v = (vector float *)pos;
     ctx.vel_v = (vector float *)vel;
     ctx.mass = mass;
     ctx.force_v = *((vector float *)&force);
     ctx.dt = dt;
```

```
    spe_id = spe_create_thread(0, &particle, &ctx, NULL, -1, 0);
    if (spe_id)
        (void)spe_wait(spe_id, &status, 0);
    else{
        perror("Unable to create SPE thread");

        return (1);
    }

    return (0);
}
```

particle.c

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "particle.h"
#include <sum_across_float4.h>
#include <math.h>

#define PARTICLES_PER_BLOCK                    500

volatile context ctx;
volatile vector float pos_i[PARTICLES_PER_BLOCK];
volatile vector float vel_i[PARTICLES_PER_BLOCK];
volatile float mass_i[PARTICLES_PER_BLOCK];
volatile vector float pos_j[PARTICLES_PER_BLOCK];
volatile vector float vel_j[PARTICLES_PER_BLOCK];
volatile float mass_j[PARTICLES_PER_BLOCK];

vector float reciprocal(vector float a)
{
    vector float one = spu_splats(1.0f);
    vector float estimate= spu_re(a);

    return spu_madd( spu_nmsub( estimate, a, one ), estimate, estimate );
}

vector float divide(vector float a, vector float b)
{
    return spu_mul(a, reciprocal(b));
}

int main(unsigned long long spu_id, unsigned long long parm)
{
    int i, j, k, l;
    int step = 0;
    unsigned int tag_id = 0;
    float time;
    vector float dt_v;
    vector float zero = spu_splats(0.0f);
    vector float temp = zero;

    spu_writech(MFC_WrTagMask, -1);
    spu_mfcdma32((void *)(&ctx), (unsigned int)parm, sizeof(context), tag_id, MFC_GET_CMD);
    (void)spu_mfcstat(2);
    dt_v = spu_splats(ctx.dt);
```

```
for (time=0; time<END_OF_TIME; time += ctx.dt) {
      for (i = 0; i<ctx.particles; i+=PARTICLES_PER_BLOCK) {
            spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GETB_CMD);
            spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GET_CMD);
            spu_mfcdma32((void *)(mass_i), (unsigned int)(ctx.mass+i), PARTICLES_PER_BLOCK *
                        sizeof(float), tag_id, MFC_GET_CMD);
            (void)spu_mfcstat(2); //load i block
            for(j = i; j < ctx.particles; j+=PARTICLES_PER_BLOCK){
                  spu_mfcdma32((void *)(pos_j), (unsigned int)(ctx.pos_v+j), PARTICLES_PER_BLOCK
                              * sizeof(vector float), tag_id, MFC_GETB_CMD);
                  spu_mfcdma32((void *)(vel_j), (unsigned int)(ctx.vel_v+j), PARTICLES_PER_BLOCK
                              *sizeof(vector float), tag_id, MFC_GET_CMD);
                  spu_mfcdma32((void *)(mass_j), (unsigned int)(ctx.mass+j), PARTICLES_PER_BLOCK
                              * sizeof(float), tag_id, MFC_GET_CMD);
                  (void)spu_mfcstat(2); //load j block
                  for(k = 0; k < PARTICLES_PER_BLOCK; k++){
                        l = 0;
                        if(j==1) l=k+1;
                        for(; l < PARTICLES_PER_BLOCK; l++){
                              if(i != j || l !=k){
                                    vector float forceCalc = spu_sub(pos_i[k],pos_j[l]);
                                    temp = spu_mul(forceCalc, forceCalc);
                                    float dist __attribute__ ((aligned (16)))= sqrt(_sum_across_float4(temp));
                                    vector float dist_v = spu_splats(dist);
                                    vector float mag = divide(dt_v, spu_mul(spu_mul(dist_v, dist_v), dist_v));
                                    vector float mass_vi = spu_splats(mass_i[k]);
                                    vector float mass_vj = spu_splats(mass_j[l]);

                                    vel_j[l] = spu_madd(spu_mul(mass_vi, forceCalc), mag ,vel_j[l]);
                                    vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj, forceCalc), mag));
                              }
                        }
                  }
                  spu_mfcdma32((void *)(vel_j), (unsigned int)(ctx.vel_v+j), PARTICLES_PER_BLOCK *
                              sizeof(vector float), tag_id, MFC_PUT_CMD); //store j block
            }
            spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_PUT_CMD); //store i block
      }
      for (i = 0; i<ctx.particles; i+=PARTICLES_PER_BLOCK) {
            spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GETB_CMD);
            spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GET_CMD);
            (void)spu_mfcstat(2); //load i block
            for( k = 0; k < PARTICLES_PER_BLOCK; k++)
                  pos_i[k] = spu_madd(vel_i[k], dt_v, pos_i[k]);
            spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_PUT_CMD); //store i block
      }
}
(void)spu_mfcstat(2);

return (0);
```

```
}
```

Step 4b: Leap Single SPE (One-Way Interaction)

leap.h
See Step 4a.

particle.h
See Step 4a.

euler_spe.c
See Step 4a.

particle.c

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "particle.h"
#include <sum_across_float4.h>
#include <math.h>

#define PARTICLES_PER_BLOCK                    500

volatile context ctx;
volatile vector float pos_i[PARTICLES_PER_BLOCK];
volatile vector float vel_i[PARTICLES_PER_BLOCK];
volatile float mass_i[PARTICLES_PER_BLOCK];
volatile vector float pos_j[PARTICLES_PER_BLOCK];
volatile float mass_j[PARTICLES_PER_BLOCK];

vector float reciprocal(vector float a)
{
    vector float one = spu_splats(1.0f);
    vector float estimate= spu_re(a);

    return spu_madd( spu_nmsub( estimate, a, one ), estimate, estimate );
}

vector float divide(vector float a, vector float b)
{
    return spu_mul(a, reciprocal(b));
}

int main(unsigned long long spu_id, unsigned long long parm)
{
    int i, j, k, l;
    int step = 0;
    unsigned int tag_id = 0;
    float time;
    vector float dt_v;
    vector float zero = spu_splats(0.0f);
    vector float temp = zero;
```

```
    spu_writech(MFC_WrTagMask, -1);
    spu_mfcdma32((void *)(&ctx), (unsigned int)parm, sizeof(context), tag_id, MFC_GET_CMD);
    (void)spu_mfcstat(2);
    dt_v = spu_splats(ctx.dt);
    for (time=0; time<END_OF_TIME; time += ctx.dt) {
        for (i = 0; i<ctx.particles; i+=PARTICLES_PER_BLOCK){
            spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GETB_CMD);
            spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GET_CMD);
            spu_mfcdma32((void *)(mass_i), (unsigned int)(ctx.mass+i), PARTICLES_PER_BLOCK *
                       sizeof(float), tag_id, MFC_GET_CMD);
            (void)spu_mfcstat(2); //load i block
            for(j = 0; j < ctx.particles; j+=PARTICLES_PER_BLOCK){
                spu_mfcdma32((void *)(pos_j), (unsigned int)(ctx.pos_v+j), PARTICLES_PER_BLOCK
                            * sizeof(vector float), tag_id, MFC_GETB_CMD);
                spu_mfcdma32((void *)(mass_j), (unsigned int)(ctx.mass+j),
                            PARTICLES_PER_BLOCK * sizeof(float), tag_id, MFC_GET_CMD);
                (void)spu_mfcstat(2); //load j block

            for(k = 0; k < PARTICLES_PER_BLOCK; k++){
                l = 0;
                for(; l < PARTICLES_PER_BLOCK; l++){
                    if(i != j || l !=k){
                        vector float forceCalc = spu_sub(pos_i[k],pos_j[l]);
                        temp = spu_mul(forceCalc, forceCalc);
                        float dist __attribute__ ((aligned (16)))= sqrt(_sum_across_float4(temp));
                        vector float dist_v = spu_splats(dist);
                        vector float mag = divide(dt_v, spu_mul(spu_mul(dist_v, dist_v), dist_v));
                        vector float mass_vj = spu_splats(mass_j[l]);

                        vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj, forceCalc), mag));
                    }
                }
            }
        }
        for( k = 0; k < PARTICLES_PER_BLOCK; k++)
            pos_i[k] = spu_madd(vel_i[k], dt_v, pos_i[k]);

        spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), tag_id, MFC_PUT_CMD);
        spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), tag_id, MFC_PUT_CMD); //store i block
        }
    }
    (void)spu_mfcstat(2);

    return (0);
}
```

Step 5a: Leap Multiple SPE (Default)

leap.h

```
#ifndef _LEAP_H_
```

```
#define _LEAP_H_

#define END_OF_TIME   6.28
#define PARTICLES        10000
#define PARTICLES_PER_BLOCK 500

typedef struct {
     float x, y, z, w;
} vec4D;

#endif /* _LEAP_H_ */
```

particle.h

```
#include "leap.h"

typedef struct {
     int particles;  // number of particle to process
     vector float *pos_v;    // pointer to array of positions vectors
     vector float *vel_v;    // pointer to array of velocity vectors
     float *mass;  // pointer to array of mass vectors
     vector float force_v;   // force vector
     float dt;                    // current step in time
     int startBlock, endBlock;
} context;
```

euler_spe.c

```
#include <stdio.h>
#include <libspe.h>
#include "particle.h"

#define SPE_THREADS              7

vec4D pos[PARTICLES] __attribute__ ((aligned (16)));
vec4D vel[PARTICLES] __attribute__ ((aligned (16)));
vec4D force __attribute__ ((aligned (16)));
float mass[PARTICLES] __attribute__ ((aligned (16)));
float dt = 0.01f;

extern spe_program_handle_t particle;

int main()
{
      int status, i, numBlocks, blocksPerThread, extras, block;
      float time;
      speid_t spe_ids [SPE_THREADS];
      context ctxs[SPE_THREADS] __attribute__ ((aligned (16)));

      numBlocks = PARTICLES/PARTICLES_PER_BLOCK;
      blocksPerThread = numBlocks/SPE_THREADS;
      extras = numBlocks - (blocksPerThread * SPE_THREADS);
      for(i = 0; i < PARTICLES; i++){
```

```
            pos[i].x = 0.0f;
            vel[i].x = 0.0f;
            pos[i].y = 0.0f;
            vel[i].y = 0.0f;
            pos[i].z = 0.0f;
            vel[i].z = 0.0f;
            pos[i].w = 0.0f;
            vel[i].w = 0.0f;
            mass[i] = 1.0f;
        }
        for(i = 1; i < PARTICLES; i++){
            pos[i].x = (float) i;
            vel[i].y = 1/sqrt((float) i);
            mass[i] = 1e-4f;
        }
        for (time=0; time<END_OF_TIME; time += dt){
            block = 0;

            for (i=0; i<SPE_THREADS; i++){
                ctxs[i].particles = PARTICLES;
                ctxs[i].pos_v = (vector float *)pos;
                ctxs[i].vel_v = (vector float *)vel;
                ctxs[i].mass = mass;
                ctxs[i].force_v = *((vector float *)&force);
                ctxs[i].dt = dt;
                ctxs[i].startBlock = block;
                block+=blocksPerThread;
                if(i < extras)
                    block++;
                ctxs[i].endBlock = block;
                spe_ids[i] = spe_create_thread(0, &particle, &ctxs[i], NULL, -1, 0);
                if (spe_ids[i] == -1) {
                    perror("Unable to create SPE thread");

                    return (1);

                }
            }
            for (i=0; i<SPE_THREADS; i++) {
                (void)spe_wait(spe_ids[i], &status, 0);
            }
        }

        return (0);
}
```

particle.c

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "particle.h"
#include <sum_across_float4.h>
#include <math.h>

volatile context ctx;
```

```
volatile vector float pos_i[PARTICLES_PER_BLOCK];
volatile vector float vel_i[PARTICLES_PER_BLOCK];
volatile float mass_i[PARTICLES_PER_BLOCK];
volatile vector float pos_j[PARTICLES_PER_BLOCK];
volatile vector float vel_j[PARTICLES_PER_BLOCK];
volatile float mass_j[PARTICLES_PER_BLOCK];

vector float reciprocal(vector float a)
{
    vector float one = spu_splats(1.0f);
    vector float estimate= spu_re(a);

    return spu_madd( spu_nmsub( estimate, a, one ), estimate, estimate );
}

vector float divide(vector float a, vector float b)
{
    return spu_mul(a, reciprocal(b));
}

int main(unsigned long long spu_id, unsigned long long parm)
{
    int i, j, k, l;
    int step = 0;
    unsigned int tag_id = 0;
    float time;
    vector float dt_v;
    vector float zero = spu_splats(0.0f);
    vector float temp = zero;

    spu_writech(MFC_WrTagMask, -1);
    spu_mfcdma32((void *)(&ctx), (unsigned int)parm, sizeof(context), tag_id, MFC_GET_CMD);
    (void)spu_mfcstat(2);
    dt_v = spu_splats(ctx.dt);
    for (i = ctx.startBlock * PARTICLES_PER_BLOCK; i<ctx.endBlock * PARTICLES_PER_BLOCK;
        i+=PARTICLES_PER_BLOCK){
        spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), tag_id, MFC_GETB_CMD);
        spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), tag_id, MFC_GET_CMD);
        spu_mfcdma32((void *)(mass_i), (unsigned int)(ctx.mass+i), PARTICLES_PER_BLOCK *
                    sizeof(float), tag_id, MFC_GET_CMD);
        (void)spu_mfcstat(2); //load i block
        for(j = 0; j < ctx.particles; j+=PARTICLES_PER_BLOCK){
            spu_mfcdma32((void *)(pos_j), (unsigned int)(ctx.pos_v+j), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GETB_CMD);
            spu_mfcdma32((void *)(vel_j), (unsigned int)(ctx.vel_v+j), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GET_CMD);
            spu_mfcdma32((void *)(mass_j), (unsigned int)(ctx.mass+j), PARTICLES_PER_BLOCK *
                        sizeof(float), tag_id, MFC_GET_CMD);
            (void)spu_mfcstat(2); //load j block
            for(k = 0; k < PARTICLES_PER_BLOCK; k++){
                l = 0;
                for(; l < PARTICLES_PER_BLOCK; l++){
                    if(i != j || l !=k){
                        vector float forceCalc = spu_sub(pos_i[k],pos_j[l]);
```

```
                        temp = spu_mul(forceCalc, forceCalc);
                        float dist __attribute__ ((aligned (16)))= sqrt(_sum_across_float4(temp));
                        vector float dist_v = spu_splats(dist);
                        vector float mag = divide(dt_v, spu_mul(spu_mul(dist_v, dist_v), dist_v));
                        vector float mass_vj = spu_splats(mass_j[l]);

                        vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj, forceCalc), mag));
                    }
                }
            }
        }
        for( k = 0; k < PARTICLES_PER_BLOCK; k++)
            pos_i[k] = spu_madd(vel_i[k], dt_v, pos_i[k]);

        spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_PUT_CMD);
        spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_PUT_CMD); //store i block
    }
    (void)spu_mfcstat(2);

    return (0);
}
```

Step 5b: Leap Multiple SPE (Double Buffered)

leap.h
See Step 5a.

particle.h
See Step 5a.

euler_spe.c
See Step 5a.

particle.c

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "particle.h"
#include <sum_across_float4.h>
#include <math.h>

volatile context ctx;
volatile vector float pos_i[PARTICLES_PER_BLOCK];
volatile vector float vel_i[PARTICLES_PER_BLOCK];
volatile float mass_i[PARTICLES_PER_BLOCK];
volatile vector float pos_j[2][PARTICLES_PER_BLOCK];
volatile float mass_j[2][PARTICLES_PER_BLOCK];

vector float reciprocal(vector float a)
{
    vector float one = spu_splats(1.0f);
```

```
    vector float estimate= spu_re(a);

    return spu_madd( spu_nmsub( estimate, a, one ), estimate, estimate );
}

vector float divide(vector float a, vector float b)
{
    return spu_mul(a, reciprocal(b));
}

void process_buffer(int buffer, vector float dt_v, int i, int j)
{
    int k, l;
    vector float temp = spu_splats(0.0f);

    for(k = 0; k < PARTICLES_PER_BLOCK; k++){
        l = 0;
        for(; l < PARTICLES_PER_BLOCK; l++){
            if(i != j || l !=k){
                vector float forceCalc = spu_sub(pos_i[k],pos_j[buffer][l]);
                temp = spu_mul(forceCalc, forceCalc);
                float dist __attribute__ ((aligned (16)))= sqrt(_sum_across_float4(temp));
                vector float dist_v = spu_splats(dist);
                vector float mag = divide(dt_v, spu_mul(spu_mul(dist_v, dist_v), dist_v));
                vector float mass_vj = spu_splats(mass_j[buffer][l]);

                vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj, forceCalc), mag));
            }
        }
    }
}

int main(unsigned long long spu_id, unsigned long long parm)
{
    int i, j, k;
    int buffer, next_buffer;
    vector float dt_v;
    volatile vector float *ctx_pos_v;
    volatile vector float *next_ctx_pos_v;
    volatile float *ctx_mass, *next_ctx_mass;

    spu_writech(MFC_WrTagMask, 1 << 0);
    spu_mfcdma32((void *)(&ctx), (unsigned int)parm, sizeof(context), 0, MFC_GET_CMD);
    (void)spu_mfcstat(2);
    dt_v = spu_splats(ctx.dt);
    for (i = ctx.startBlock * PARTICLES_PER_BLOCK; i<ctx.endBlock * PARTICLES_PER_BLOCK;
        i+=PARTICLES_PER_BLOCK){
        ctx_pos_v = ctx.pos_v;
        ctx_mass = ctx.mass;
    spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), 0, MFC_GETB_CMD);
    spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), 0, MFC_GET_CMD);
    spu_mfcdma32((void *)(mass_i), (unsigned int)(ctx.mass+i), PARTICLES_PER_BLOCK *
                    sizeof(float), 0, MFC_GET_CMD);
    (void)spu_mfcstat(2); //load i block
```

```
    buffer = 0;
    spu_mfcdma32((void *)(pos_j), (unsigned int)(ctx_pos_v), PARTICLES_PER_BLOCK *
                    sizeof(vector float), 0, MFC_GETB_CMD);
    spu_mfcdma32((void *)(mass_j), (unsigned int)(ctx_mass), PARTICLES_PER_BLOCK *
                    sizeof(float), 0, MFC_GET_CMD);
    for(j = 0; j < ctx.particles; j+=PARTICLES_PER_BLOCK){
        if(j < ctx.particles - PARTICLES_PER_BLOCK){
            next_ctx_pos_v = ctx_pos_v + PARTICLES_PER_BLOCK;
            next_ctx_mass = ctx_mass + PARTICLES_PER_BLOCK;
            next_buffer = buffer^1;
            spu_mfcdma32((void *)(&pos_j[next_buffer][0]), (unsigned int)(next_ctx_pos_v),
                            PARTICLES_PER_BLOCK * sizeof(vector float), next_buffer,
                            MFC_GETB_CMD);
            spu_mfcdma32((void *)(&mass_j[next_buffer][0]), (unsigned int)(next_ctx_mass),
                           PARTICLES_PER_BLOCK * sizeof(float), next_buffer, MFC_GET_CMD);
        }
        spu_writech(MFC_WrTagMask, 1 << buffer);
        (void)spu_mfcstat(2);
        process_buffer(buffer, dt_v, i, j);
        ctx_pos_v = next_ctx_pos_v;
        ctx_mass = next_ctx_mass;
        buffer = next_buffer;
    }
    for( k = 0; k < PARTICLES_PER_BLOCK; k++)
        pos_i[k] = spu_madd(vel_i[k], dt_v, pos_i[k]);
    spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), 0, MFC_PUT_CMD);
    spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), 0, MFC_PUT_CMD); //store i block
 }

 (void)spu_mfcstat(2);

 return (0);
}
```

Step 5c: Leap Multiple SPE (Unrolled Loop)

leap.h
See Step 5a.

particle.h
See Step 5a.

euler_spe.c
See Step 5a.

particle.c

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "particle.h"
#include <sum_across_float4.h>
```

```
#include <math.h>

// Local store structures and buffers.
volatile context ctx;
volatile vector float pos_i[PARTICLES_PER_BLOCK];
volatile vector float vel_i[PARTICLES_PER_BLOCK];
volatile float mass_i[PARTICLES_PER_BLOCK];
volatile vector float pos_j[PARTICLES_PER_BLOCK];
volatile vector float vel_j[PARTICLES_PER_BLOCK];
volatile float mass_j[PARTICLES_PER_BLOCK];

vector float reciprocal(vector float a)
{
    vector float one = spu_splats(1.0f);
    vector float estimate= spu_re(a);

    return spu_madd( spu_nmsub( estimate, a, one ), estimate, estimate );
}

vector float divide(vector float a, vector float b)
{
    return spu_mul(a, reciprocal(b));
}

void process_buffer(vector float dt_v, int i, int j)
{
     int k, l, m;
     vector float forceCalc0, forceCalc1, forceCalc2, forceCalc3;
     vector float temp0, temp1, temp2, temp3;
     vector float dist_v0, dist_v1, dist_v2, dist_v3;
     vector float mag0, mag1, mag2, mag3;
     vector float mass_vj0, mass_vj1, mass_vj2, mass_vj3;
     float dist0 __attribute__ ((aligned (16))), dist1 __attribute__ ((aligned (16))), dist2 __attribute__
     ((aligned (16))), dist3 __attribute__ ((aligned (16)));

     for(k = 0; k < PARTICLES_PER_BLOCK; k++){
          for(l = 0; l < PARTICLES_PER_BLOCK; l+=4){

               if(i != j || k < l || k > l+3){
                    forceCalc0 = spu_sub(pos_i[k],pos_j[l]);
                    forceCalc1 = spu_sub(pos_i[k],pos_j[l+1]);
                    forceCalc2 = spu_sub(pos_i[k],pos_j[l+2]);
                    forceCalc3 = spu_sub(pos_i[k],pos_j[l+3]);
                    temp0 = spu_mul(forceCalc0, forceCalc0);
                    temp1 = spu_mul(forceCalc1, forceCalc1);
                    temp2 = spu_mul(forceCalc2, forceCalc2);
                    temp3 = spu_mul(forceCalc3, forceCalc3);
                    dist0 = sqrt(_sum_across_float4(temp0));
                    dist1 = sqrt(_sum_across_float4(temp1));
                    dist2 = sqrt(_sum_across_float4(temp2));
                    dist3 = sqrt(_sum_across_float4(temp3));
                    dist_v0 = spu_splats(dist0);
                    dist_v1 = spu_splats(dist1);
                    dist_v2 = spu_splats(dist2);
                    dist_v3 = spu_splats(dist3);
                     mag0 = divide(dt_v, spu_mul(spu_mul(dist_v0, dist_v0), dist_v0));
```

```
                    mag1 = divide(dt_v, spu_mul(spu_mul(dist_v1, dist_v1), dist_v1));
                    mag2 = divide(dt_v, spu_mul(spu_mul(dist_v2, dist_v2), dist_v2));
                    mag3 = divide(dt_v, spu_mul(spu_mul(dist_v3, dist_v3), dist_v3));
                    mass_vj0 = spu_splats(mass_j[l]);
                    mass_vj1 = spu_splats(mass_j[l+1]);
                    mass_vj2 = spu_splats(mass_j[l+2]);
                    mass_vj3 = spu_splats(mass_j[l+3]);
                    vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj0, forceCalc0), mag0));
                    vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj1, forceCalc1), mag1));
                    vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj2, forceCalc2), mag2));
                    vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj3, forceCalc3), mag3));
                }
                else{
                    for(m = 0; m < 4; m++){
                        if(k != l+m){
                            forceCalc0 = spu_sub(pos_i[k],pos_j[l+m]);
                            temp0 = spu_mul(forceCalc0, forceCalc0);
                            dist0= sqrt(_sum_across_float4(temp0));
                            dist_v0 = spu_splats(dist0);
                            mag0 = divide(dt_v, spu_mul(spu_mul(dist_v0, dist_v0), dist_v0));
                            mass_vj0 = spu_splats(mass_j[l+m]);
                            vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj0, forceCalc0), mag0));
                            }
                    }
                }
            }
        }
}

int main(unsigned long long spu_id, unsigned long long parm)
{
    int i, j, k, l;
    int step = 0;
    unsigned int tag_id = 0;
    float time;
    vector float dt_v;
    vector float zero = spu_splats(0.0f);
    vector float temp = zero;

    spu_writech(MFC_WrTagMask, -1);
    spu_mfcdma32((void *)(&ctx), (unsigned int)parm, sizeof(context), tag_id, MFC_GET_CMD);
    (void)spu_mfcstat(2);
    dt_v = spu_splats(ctx.dt);
    for (i = ctx.startBlock * PARTICLES_PER_BLOCK; i<ctx.endBlock * PARTICLES_PER_BLOCK;
        i+=PARTICLES_PER_BLOCK) {
        spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GETB_CMD);
        spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GET_CMD);
        spu_mfcdma32((void *)(mass_i), (unsigned int)(ctx.mass+i), PARTICLES_PER_BLOCK *
                        sizeof(float), tag_id, MFC_GET_CMD);
        (void)spu_mfcstat(2); //load i block
        for(j = 0; j < ctx.particles; j+=PARTICLES_PER_BLOCK){
            spu_mfcdma32((void *)(pos_j), (unsigned int)(ctx.pos_v+j), PARTICLES_PER_BLOCK *
                            sizeof(vector float), tag_id, MFC_GETB_CMD);
```

```
            spu_mfcdma32((void *)(vel_j), (unsigned int)(ctx.vel_v+j), PARTICLES_PER_BLOCK *
                        sizeof(vector float), tag_id, MFC_GET_CMD);
            spu_mfcdma32((void *)(mass_j), (unsigned int)(ctx.mass+j), PARTICLES_PER_BLOCK *
                        sizeof(float), tag_id, MFC_GET_CMD);
            (void)spu_mfcstat(2); //load j block
            process_buffer(dt_v, i, j);
        }
        for( k = 0; k < PARTICLES_PER_BLOCK; k++)
            pos_i[k] = spu_madd(vel_i[k], dt_v, pos_i[k]);
        spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), tag_id, MFC_PUT_CMD);
        spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), tag_id, MFC_PUT_CMD); //store i block
    }
    (void)spu_mfcstat(2);

    return (0);
}
```

Step 5d: Leap Multiple SPE (Complete)

leap.h
See Step 5a.

particle.h
See Step 5a.

euler_spe.c
See Step 5a.

particle.c

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "particle.h"
#include <sum_across_float4.h>
#include <math.h>

volatile context ctx;
volatile vector float pos_i[PARTICLES_PER_BLOCK];
volatile vector float vel_i[PARTICLES_PER_BLOCK];
volatile float mass_i[PARTICLES_PER_BLOCK];
volatile vector float pos_j[2][PARTICLES_PER_BLOCK];
volatile float mass_j[2][PARTICLES_PER_BLOCK];

vector float reciprocal(vector float a)
{
    vector float one = spu_splats(1.0f);
    vector float estimate= spu_re(a);

    return spu_madd( spu_nmsub( estimate, a, one ), estimate, estimate );
}
```

```
vector float divide(vector float a, vector float b)
{
    return spu_mul(a, reciprocal(b));
}

void process_buffer(int buffer, vector float dt_v, int i, int j)
{
    int k, l, m;
    vector float forceCalc0, forceCalc1, forceCalc2, forceCalc3;
    vector float temp0, temp1, temp2, temp3;
    vector float dist_v0, dist_v1, dist_v2, dist_v3;
    vector float mag0, mag1, mag2, mag3;
    vector float mass_vj0, mass_vj1, mass_vj2, mass_vj3;
    float dist0 __attribute__ ((aligned (16))), dist1 __attribute__ ((aligned (16))), dist2 __attribute__
        ((aligned (16))), dist3 __attribute__ ((aligned (16)));

    for(k = 0; k < PARTICLES_PER_BLOCK; k++){
        for(l = 0; l < PARTICLES_PER_BLOCK; l+=4){
            if(i != j || k < l || k > l+3){
                forceCalc0 = spu_sub(pos_i[k],pos_j[buffer][l]);
                forceCalc1 = spu_sub(pos_i[k],pos_j[buffer][l+1]);
                forceCalc2 = spu_sub(pos_i[k],pos_j[buffer][l+2]);
                forceCalc3 = spu_sub(pos_i[k],pos_j[buffer][l+3]);
                temp0 = spu_mul(forceCalc0, forceCalc0);
                temp1 = spu_mul(forceCalc1, forceCalc1);
                temp2 = spu_mul(forceCalc2, forceCalc2);
                temp3 = spu_mul(forceCalc3, forceCalc3);
                dist0 = sqrt(_sum_across_float4(temp0));
                dist1 = sqrt(_sum_across_float4(temp1));
                dist2 = sqrt(_sum_across_float4(temp2));
                dist3 = sqrt(_sum_across_float4(temp3));
                dist_v0 = spu_splats(dist0);
                dist_v1 = spu_splats(dist1);
                dist_v2 = spu_splats(dist2);
                dist_v3 = spu_splats(dist3);
                mag0 = divide(dt_v, spu_mul(spu_mul(dist_v0, dist_v0), dist_v0));
                mag1 = divide(dt_v, spu_mul(spu_mul(dist_v1, dist_v1), dist_v1));
                mag2 = divide(dt_v, spu_mul(spu_mul(dist_v2, dist_v2), dist_v2));
                mag3 = divide(dt_v, spu_mul(spu_mul(dist_v3, dist_v3), dist_v3));
                mass_vj0 = spu_splats(mass_j[buffer][l]);
                mass_vj1 = spu_splats(mass_j[buffer][l+1]);
                mass_vj2 = spu_splats(mass_j[buffer][l+2]);
                mass_vj3 = spu_splats(mass_j[buffer][l+3]);
                vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj0, forceCalc0), mag0));
                vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj1, forceCalc1), mag1));
                vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj2, forceCalc2), mag2));
                vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj3, forceCalc3), mag3));
            }
            else{
                for(m = 0; m < 4; m++){
                    if(k != l+m){
                        forceCalc0 = spu_sub(pos_i[k],pos_j[buffer][l+m]);
                        temp0 = spu_mul(forceCalc0, forceCalc0);
                        dist0= sqrt(_sum_across_float4(temp0));
                        dist_v0 = spu_splats(dist0);
                        mag0 = divide(dt_v, spu_mul(spu_mul(dist_v0, dist_v0), dist_v0));
```

```
                        mass_vj0 = spu_splats(mass_j[buffer][l+m]);
                        vel_i[k] = spu_sub(vel_i[k], spu_mul(spu_mul(mass_vj0, forceCalc0), mag0));
                    }
                }
            }
        }
    }
}

int main(unsigned long long spu_id, unsigned long long parm)
{
    int i, j, k;
    int buffer, next_buffer;
    vector float dt_v;
    volatile vector float *ctx_pos_v;
    volatile vector float *next_ctx_pos_v;
    volatile float *ctx_mass, *next_ctx_mass;

    spu_writech(MFC_WrTagMask, 1 << 0);
    spu_mfcdma32((void *)(&ctx), (unsigned int)parm, sizeof(context), 0, MFC_GET_CMD);
    (void)spu_mfcstat(2);
    dt_v = spu_splats(ctx.dt);
    for (i = ctx.startBlock * PARTICLES_PER_BLOCK; i<ctx.endBlock * PARTICLES_PER_BLOCK;
        i+=PARTICLES_PER_BLOCK){
        ctx_pos_v = ctx.pos_v;
        ctx_mass = ctx.mass;
        spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), 0, MFC_GETB_CMD);
        spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                    sizeof(vector float), 0, MFC_GET_CMD);
        spu_mfcdma32((void *)(mass_i), (unsigned int)(ctx.mass+i), PARTICLES_PER_BLOCK *
                    sizeof(float), 0, MFC_GET_CMD);
        (void)spu_mfcstat(2); //load i block
        buffer = 0;
        spu_mfcdma32((void *)(pos_j), (unsigned int)(ctx_pos_v), PARTICLES_PER_BLOCK *
                    sizeof(vector float), 0, MFC_GETB_CMD);
        spu_mfcdma32((void *)(mass_j), (unsigned int)(ctx_mass), PARTICLES_PER_BLOCK *
                    sizeof(float), 0, MFC_GET_CMD);
        for(j = 0; j < ctx.particles; j+=PARTICLES_PER_BLOCK){
            if(j < ctx.particles - PARTICLES_PER_BLOCK){
                next_ctx_pos_v = ctx_pos_v + PARTICLES_PER_BLOCK;
                next_ctx_mass = ctx_mass + PARTICLES_PER_BLOCK;
                next_buffer = buffer^1;
                spu_mfcdma32((void *)(&pos_j[next_buffer][0]), (unsigned int)(next_ctx_pos_v),
                            PARTICLES_PER_BLOCK * sizeof(vector float), next_buffer,
                            MFC_GETB_CMD);
                spu_mfcdma32((void *)(&mass_j[next_buffer][0]), (unsigned int)(next_ctx_mass),
                            PARTICLES_PER_BLOCK * sizeof(float), next_buffer,
                            MFC_GET_CMD);
            }
            spu_writech(MFC_WrTagMask, 1 << buffer);
            (void)spu_mfcstat(2);
            process_buffer(buffer, dt_v, i, j);
            ctx_pos_v = next_ctx_pos_v;
            ctx_mass = next_ctx_mass;
            buffer = next_buffer;
```

```
            }
            for( k = 0; k < PARTICLES_PER_BLOCK; k++)
                pos_i[k] = spu_madd(vel_i[k], dt_v, pos_i[k]);;
            spu_mfcdma32((void *)(pos_i), (unsigned int)(ctx.pos_v+i), PARTICLES_PER_BLOCK *
                                sizeof(vector float), 0, MFC_PUT_CMD);
            spu_mfcdma32((void *)(vel_i), (unsigned int)(ctx.vel_v+i), PARTICLES_PER_BLOCK *
                                sizeof(vector float), 0, MFC_PUT_CMD); //store i block
    }
    (void)spu_mfcstat(2);

    return (0);
}
```